

# CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual

Document Number: CWMCUS08ASMREF  
Rev 10.6, 02/2014



# Contents

Section number	Title	Page
<b>Chapter 1</b>		
<b>Using HC(S)08/RS08 Assembler</b>		
1.1	Highlights.....	25
1.2	Structure of this Document.....	25
<b>Chapter 2</b>		
<b>Working with Assembler</b>		
2.1	Programming Overview.....	27
2.1.1	Project Directory.....	29
2.1.2	External Editor.....	29
2.2	Managing Assembly Language Project Using CodeWarrior IDE.....	30
2.2.1	Create New Project.....	30
2.2.2	Additional Project Information.....	32
2.3	Analysis of Groups and Files in a Project.....	35
2.3.1	CodeWarrior Groups.....	36
2.3.2	Creating New Group.....	37
2.3.3	Adding New File to the Project.....	37
2.3.4	Renaming File or Group.....	38
2.3.5	Moving File.....	38
2.3.6	Removing File.....	38
2.3.7	Restoring Deleted File.....	39
2.3.8	Using Editor.....	39
2.3.9	Generating Listing Files.....	40
2.4	Writing your Assembly Source Files.....	41
2.5	Analyzing Project Files.....	42
2.6	Assembling Source Files.....	46
2.6.1	Assembling and Linking with CodeWarrior IDE.....	46
2.6.2	Assembling with Assembler.....	47
2.6.2.1	Configuring Assembler.....	48

Section number	Title	Page
2.6.2.2	Input Files.....	52
2.6.2.3	Assembling Assembly Source-code Files.....	54
2.7	Linking Application.....	60
2.7.1	Linking with CodeWarrior IDE.....	60
2.7.1.1	PRM File.....	60
2.7.1.2	Linking Object-code Files.....	62
2.7.2	Linking with Linker.....	64
2.8	Directly Generating ABS File.....	69
2.8.1	Creating Absolute Assembly Project.....	69
2.8.2	Adapting Absolute Assembly File Created by Wizard.....	70
2.8.3	Generating Absolute Assembly Using CodeWarrior IDE .....	74
2.8.4	Generating Absolute Assembly Using Assembler Build Tool.....	74
2.9	Assembler Build Properties Panels.....	75
2.9.1	HCS08 Assembler Build Properties Panels.....	76
2.9.1.1	HCS08 Assembler.....	77
2.9.1.2	HCS08 Assembler > Output.....	77
2.9.1.3	HCS08 Assembler > Output > Configure listing file.....	79
2.9.1.4	HCS08 Assembler > Input.....	79
2.9.1.5	HCS08 Assembler > Language.....	80
2.9.1.6	HCS08 Assembler > Language > Compatibility modes.....	81
2.9.1.7	HCS08 Assembler > Host.....	83
2.9.1.8	HCS08 Assembler > Code Generation.....	83
2.9.1.9	HCS08 Assembler > Messages.....	84
2.9.1.10	HCS08 Assembler > Messages > Disable user messages.....	85
2.9.1.11	HCS08 Assembler > General.....	86
2.9.2	RS08 Assembler Build Properties Panels.....	87
2.9.2.1	RS08 Assembler.....	87
2.9.2.2	RS08 Assembler > Output.....	88
2.9.2.3	RS08 Assembler > Output > Configure Listing File.....	89

Section number	Title	Page
2.9.2.4	RS08 Assembler > Input.....	90
2.9.2.5	RS08 Assembler > Language.....	92
2.9.2.6	RS08 Assembler > Language > Compatibility modes.....	93
2.9.2.7	RS08 Assembler > Host.....	94
2.9.2.8	RS08 Assembler > Code Generation.....	95
2.9.2.9	RS08 Assembler > Messages.....	95
2.9.2.10	RS08 Assembler > Messages > Disable user messages.....	97
2.9.2.11	RS08 Assembler > General.....	98

### Chapter 3 Assembler Graphical User Interface

3.1	Starting Assembler.....	99
3.2	Assembler Main Window.....	100
3.2.1	Window Title.....	100
3.2.2	Content Area.....	101
3.2.3	Toolbar.....	102
3.2.4	Status Bar.....	102
3.2.5	Assembler Menu Bar.....	103
3.2.6	File Menu.....	103
3.2.7	Assembler Menu.....	104
3.2.8	View Menu.....	105
3.3	Editor Setting Dialog Box.....	105
3.3.1	Global Editor (shared by all tools and projects).....	105
3.3.2	Local Editor (shared by all tools).....	106
3.3.3	Editor Started with Command Line.....	107
3.3.3.1	Example of Configuring a Command Line Editor.....	108
3.3.4	Editor Started with DDE.....	108
3.3.5	CodeWarrior with COM.....	109
3.3.6	Modifiers.....	110

Section number	Title	Page
3.4	Save Configuration Dialog Box.....	111
3.4.1	Environment Configuration Dialog Box.....	113
3.5	Option Settings Dialog Box.....	114
3.6	Message Settings Dialog Box.....	115
3.6.1	Changing the Class Associated with a Message.....	117
3.6.1.1	Example.....	117
3.7	About Dialog Box.....	118
3.8	Specifying Input File.....	119
3.8.1	Use Command Line in Toolbar to Assemble.....	119
3.8.2	Assembling a New File.....	119
3.8.3	Assembling a File which has Already been Assembled.....	119
3.8.4	Use File > Assemble Entry.....	120
3.8.5	Use Drag and Drop.....	120
3.9	Message/Error Feedback.....	120
3.9.1	Use Information from Assembler Window.....	121
3.9.2	Use User-defined Editor.....	121
3.9.3	Line Number can be Specified on the Command Line.....	121
3.9.4	Line Number cannot be Specified on the Command Line.....	121

## Chapter 4 Environment

4.1	Current directory.....	124
4.2	Environment macros.....	125
4.3	Global initialization file - mctools.ini (PC only).....	126
4.4	Local configuration file (usually project.ini).....	126
4.5	Line continuation.....	128
4.6	Environment variables details.....	129
4.6.1	ABSPATH: Absolute file path.....	129
4.6.2	ASMOPTIONS: Default assembler options.....	130
4.6.3	COPYRIGHT: Copyright entry in object file.....	131

Section number	Title	Page
4.6.4	DEFAULTDIR: Default current directory.....	132
4.6.5	ENVIRONMENT: Environment file specification.....	132
4.6.6	ERRORFILE: Filename specification error.....	133
4.6.7	GENPATH: Search path for input file.....	135
4.6.8	INCLUDETIME: Creation time in the object file.....	136
4.6.9	OBJPATH: Object file path.....	137
4.6.10	SRECORD: S-Record type.....	138
4.6.11	TEXTPATH: Text file path.....	139
4.6.12	TMP: Temporary directory.....	139
4.6.13	USERNAME: User Name in object file.....	140

## Chapter 5 Files

5.1	Input files.....	143
5.1.1	Source files.....	143
5.1.2	Include files.....	143
5.2	Output files.....	144
5.2.1	Object files.....	144
5.2.2	Absolute files.....	144
5.2.3	S-Record Files.....	145
5.2.4	Listing files.....	145
5.2.5	Debug listing files.....	145
5.2.6	Error listing file.....	146
	5.2.6.1 Interactive mode (Assembler window open).....	146
	5.2.6.2 Batch mode (Assembler window not open).....	146
5.3	File processing.....	147

## Chapter 6 Assembler Options

6.1	Assembler Options.....	149
-----	------------------------	-----

Section number	Title	Page
6.2	Assembler Option details.....	149
6.2.1	Using Special Modifiers.....	150
6.2.1.1	Examples using special modifiers.....	150
6.3	List of Assembler Option.....	152
6.3.1	-ArgFile: Specify a file from which additional command line options will be read.....	154
6.3.2	-AsmDbg: Emit assembly source file information in debug sections.....	155
6.3.3	-Ci: Switch case sensitivity on label names OFF.....	155
6.3.4	-CMacAngBrack: Angle brackets for grouping Macro Arguments.....	156
6.3.5	-CMacBrackets: Square brackets for macro arguments grouping.....	157
6.3.6	-Compat: Compatibility modes.....	158
6.3.7	-CS08/-C08/-CRS08: Derivative family.....	161
6.3.8	-D: Define Label.....	162
6.3.9	-DefLabel: Improves support for data allocation directives.....	164
6.3.10	-Env: Set environment variable.....	165
6.3.11	-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format.....	166
6.3.12	-H: Short Help.....	168
6.3.13	-I: Include file path.....	169
6.3.14	-L: Generate a listing file.....	169
6.3.15	-Lasmc: Configure listing file.....	171
6.3.16	-Lasms: Configure the address size in the listing file.....	174
6.3.17	-Lc: No Macro call in listing file.....	175
6.3.18	-Ld: No macro definition in listing file.....	177
6.3.19	-Le: No Macro expansion in listing file.....	180
6.3.20	-Li: No included file in listing file.....	182
6.3.21	-Lic: License information.....	184
6.3.22	-LicA: License information about every feature in directory.....	185
6.3.23	-LicBorrow: Borrow license feature.....	186
6.3.24	-LicWait: Wait until floating license is available from floating License Server.....	187
6.3.25	-Ll: Show label statistics.....	188



Section number	Title	Page
6.3.26	-M (-Ms, -Mt): Memory model.....	189
6.3.27	-MacroNest: Configure maximum macro nesting.....	190
6.3.28	Message A1004 (available in the Online Help).....	191
6.3.29	-MCUasm: Switch compatibility with MCUasm ON.....	191
6.3.30	-MMU: Enable Memory Management Unit (MMU) Support.....	192
6.3.31	-N: Display notify box.....	192
6.3.32	-NoBeep: No beep in case of an error.....	193
6.3.33	-NoDebugInfo: No debug information for ELF/DWARF files.....	194
6.3.34	-NoEnv: Do not use environment.....	195
6.3.35	-ObjN: Object filename specification.....	196
6.3.36	-Prod: Specify project file at startup.....	197
6.3.37	-Struct: Support for structured types.....	198
6.3.38	-V: Prints the Assembler version.....	199
6.3.39	-View: Application standard occurrence.....	200
6.3.40	-W1: No information messages.....	201
6.3.41	-W2: No information and warning messages.....	201
6.3.42	-WErrFile: Create "err.log" error file.....	202
6.3.43	-Wmsg8x3: Cut filenames in Microsoft format to 8.3.....	203
6.3.44	-WmsgCE: RGB color for error messages.....	204
6.3.45	-WmsgCF: RGB color for fatal messages.....	205
6.3.46	-WmsgCI: RGB color for information messages.....	206
6.3.47	-WmsgCU: RGB color for user messages.....	206
6.3.48	-WmsgCW: RGB color for warning messages.....	207
6.3.49	-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode.....	208
6.3.50	-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode.....	210
6.3.51	-WmsgFob: Message format for batch mode.....	212
6.3.52	-WmsgFoi: Message format for interactive mode.....	213
6.3.53	-WmsgFonf: Message format for no file information.....	215
6.3.54	-WmsgFonp: Message format for no position information.....	217

Section number	Title	Page
6.3.55	-WmsgNe: Number of error messages.....	218
6.3.56	-WmsgNi: Number of Information messages.....	219
6.3.57	-WmsgNu: Disable user messages.....	220
6.3.58	-WmsgNw: Number of Warning messages.....	222
6.3.59	-WmsgSd: Setting a message to disable.....	222
6.3.60	-WmsgSe: Setting a message to Error.....	223
6.3.61	-WmsgSi: Setting a message to Information.....	224
6.3.62	-WmsgSw: Setting a Message to Warning.....	225
6.3.63	-WOutFile: Create error listing file.....	226
6.3.64	-WStdout: Write to standard output.....	227

## Chapter 7 Sections

7.1	Section attributes.....	229
7.1.1	Code sections.....	229
7.1.2	Constant sections.....	230
7.1.3	Data sections.....	230
7.2	Section types.....	230
7.2.1	Absolute sections.....	231
7.2.2	Relocatable sections.....	233
7.2.2.1	Example: Defining one RAM and one ROM area.....	233
7.2.2.2	Example: Defining multiple RAM and ROM areas.....	235
7.3	Relocatable vs. absolute sections.....	236
7.3.1	Modularity.....	236
7.3.2	Multiple developers.....	236
7.3.3	Early development.....	237
7.3.4	Enhanced portability.....	237
7.3.5	Tracking overlaps.....	237
7.3.6	Reusability.....	238

Section number	Title	Page
<b>Chapter 8</b>		
<b>Assembler Syntax</b>		
8.1	Comment line.....	239
8.2	Source line.....	239
8.2.1	Label field.....	240
8.2.2	Operation field.....	240
8.2.2.1	Instruction set.....	241
8.2.2.1.1	HC08 instruction set.....	241
8.2.2.1.2	Special HCS08 instructions.....	245
8.2.2.1.3	RS08 instruction set.....	245
8.2.2.2	Directive.....	248
8.2.2.3	Macro.....	248
8.2.3	Operand field: Addressing modes (HC(S)08).....	248
8.2.3.1	Inherent.....	249
8.2.3.2	Immediate.....	249
8.2.3.3	Direct.....	250
8.2.3.4	Extended.....	251
8.2.3.5	Indexed, no offset.....	251
8.2.3.6	Indexed, 8-bit offset.....	252
8.2.3.7	Indexed, 16-bit offset.....	253
8.2.3.8	Relative.....	254
8.2.3.9	Stack Pointer, 8-bit offset.....	254
8.2.3.10	Stack Pointer, 16-bit offset.....	255
8.2.3.11	Memory-to-memory immediate-to-direct.....	255
8.2.3.12	Memory-to-memory direct-to-direct.....	255
8.2.3.13	Memory-to-memory indexed-to-direct with post- increment.....	256
8.2.3.14	Memory-to-memory direct-to-indexed with post- increment.....	257
8.2.3.15	Indexed with post-increment.....	258
8.2.3.16	Indexed, 8-bit offset, with post-increment.....	258

Section number	Title	Page
8.2.4	Operand Field: Addressing Modes (RS08).....	259
8.2.4.1	Inherent (RS08).....	260
8.2.4.2	Tiny.....	260
8.2.4.3	Short.....	261
8.2.4.4	Direct.....	261
8.2.4.5	Extended.....	262
8.2.4.6	Relative.....	262
8.2.4.7	Immediate.....	263
8.2.4.8	Indexed.....	263
8.2.5	Comment Field.....	264
8.3	Symbols.....	264
8.3.1	User-defined symbols.....	264
8.3.2	External symbols.....	265
8.3.3	Undefined symbols.....	265
8.3.4	Reserved symbols.....	266
8.4	Constants.....	266
8.4.1	Integer constants.....	266
8.4.2	String constants.....	267
8.4.3	Floating-Point constants.....	267
8.5	Operators.....	268
8.5.1	Addition and subtraction operators (binary).....	268
8.5.2	Multiplication, division and modulo operators (binary).....	269
8.5.3	Sign operators (unary).....	269
8.5.4	Shift operators (binary).....	270
8.5.5	Bitwise operators (binary).....	271
8.5.6	Bitwise operators (unary).....	271
8.5.7	Logical operators (unary).....	272
8.5.8	Relational operators (binary).....	272
8.5.9	HIGH operator.....	273

Section number	Title	Page
8.5.10	HIGH_6_13 Operator.....	274
8.5.11	LOW operator.....	274
8.5.12	MAP_ADDR_6 Operator.....	275
8.5.13	PAGE operator.....	275
8.5.14	Force operator (unary).....	276
8.5.15	Operator precedence.....	277
8.6	Expression.....	277
8.6.1	Absolute expression.....	278
	8.6.1.1 Example.....	279
8.6.2	Simple relocatable expression.....	279
8.6.3	Unary operation result.....	280
8.6.4	Binary operations result.....	280
8.7	Translation limits.....	281

## Chapter 9 Assembler Directives

9.1	Directive Overview.....	283
9.1.1	Section-Definition directives.....	283
9.1.2	Constant-Definition directives.....	283
9.1.3	Data-Allocation directives.....	284
9.1.4	Symbol-Linkage directives.....	284
9.1.5	Assembly-Control directives.....	284
9.1.6	Listing-File Control directives.....	285
9.1.7	Macro Control directives.....	286
9.1.8	Conditional Assembly directives.....	286
9.2	Detailed descriptions of all assembler directives.....	287
9.2.1	ABSENTRY - Application entry point.....	287
9.2.2	ALIGN - Align Location Counter.....	288
9.2.3	BASE - Set number base.....	289
9.2.4	CLIST - List conditional assembly.....	290

Section number	Title	Page
9.2.5	DC - Define Constant.....	292
9.2.6	DCB - Define Constant Block.....	293
9.2.7	DS - Define Space.....	295
9.2.8	ELSE - Conditional assembly.....	296
9.2.9	END - End assembly.....	297
9.2.10	ENDFOR - End of FOR block.....	298
9.2.11	ENDIF - End conditional assembly.....	299
9.2.12	ENDM - End macro definition.....	300
9.2.13	EQU - Equate symbol value.....	300
9.2.14	EVEN - Force word alignment.....	301
9.2.15	FAIL - Generate Error message.....	302
9.2.16	FOR - Repeat assembly block.....	305
9.2.17	IF - Conditional assembly.....	307
9.2.18	IFcc - Conditional assembly.....	308
9.2.19	INCLUDE - Include text from another file.....	310
9.2.20	LIST - Enable Listing.....	310
9.2.21	LLEN - Set Line Length.....	312
9.2.22	LONGEVEN - Forcing Long-Word alignment.....	313
9.2.23	MACRO - Begin macro definition.....	314
9.2.24	MEXIT - Terminate Macro Expansion.....	315
9.2.25	MLIST - List macro expansions.....	317
9.2.26	NOLIST - Disable Listing.....	320
9.2.27	NOPAGE - Disable Paging.....	321
9.2.28	OFFSET - Create absolute symbols.....	321
9.2.29	ORG - Set Location Counter.....	323
9.2.30	PAGE - Insert Page break.....	324
9.2.31	PLEN - Set Page Length.....	325
9.2.32	RAD50 - RAD50-encoded string constants.....	326
9.2.33	SECTION - Declare Relocatable Section.....	328

Section number	Title	Page
9.2.34	SET - Set Symbol Value.....	329
9.2.35	SPC - Insert Blank Lines.....	330
9.2.36	TABS - Set Tab Length.....	331
9.2.37	TITLE - Provide Listing Title.....	331
9.2.38	XDEF - External Symbol Definition.....	332
9.2.39	XREF - External Symbol Reference.....	333
9.2.40	XREFB - External Reference for Symbols located on the Direct Page.....	333

## Chapter 10 Macros

10.1	Macro overview.....	335
10.2	Defining a macro.....	335
10.3	Calling macros.....	336
10.4	Macro parameters.....	336
10.4.1	Macro argument grouping.....	337
10.5	Labels inside macros.....	339
10.6	Macro expansion.....	340
10.7	Nested macros.....	341

## Chapter 11 Assembler Listing File

11.1	Page header.....	343
11.2	Source listing.....	344
11.2.1	Abs.....	344
11.2.2	Rel.....	346
11.2.3	Loc.....	348
11.2.4	Obj. code.....	350
11.2.5	Source line.....	351

## Chapter 12 Mixed C and Assembler Applications

12.1	Memory models.....	353
12.2	Parameter passing scheme.....	354

Section number	Title	Page
12.3	Return Value.....	354
12.4	Accessing assembly variables in an ANSI-C source file.....	354
12.5	Accessing ANSI-C variables in an assembly source file.....	355
12.6	Invoking an assembly function in an ANSI-C source file.....	356
12.6.1	Example of a C file.....	357
12.7	Support for structured types.....	359
12.7.1	Structured type definition.....	359
12.7.2	Types allowed for structured type fields.....	360
12.7.3	Variable definition.....	361
12.7.4	Variable declaration.....	361
12.7.5	Accessing a structured variable.....	362
12.7.5.1	Accessing a field address.....	362
12.7.5.2	Accessing a field offset.....	363
12.8	Structured type: Limitations.....	364

## Chapter 13 Make Applications

13.1	Assembly applications.....	365
13.1.1	Directly generating an absolute file.....	365
13.1.1.1	Generating object files.....	365
13.1.2	Mixed C and assembly applications.....	366
13.2	Memory maps and segmentation.....	366

## Chapter 14 How to...

14.1	Working with absolute sections.....	369
14.1.1	Defining absolute sections in an assembly source file.....	369
14.1.2	Linking an application containing absolute sections.....	371
14.2	Working with relocatable sections.....	372
14.2.1	Defining relocatable sections in a source file.....	372
14.2.2	Linking an application containing relocatable sections.....	373



Section number	Title	Page
14.3	Initializing Vector table.....	375
14.3.1	Initializing the Vector table in the linker PRM file.....	375
14.3.2	Initializing the Vector Table in a source file using a relocatable section.....	378
14.3.3	Initializing the Vector Table in a source file using an absolute section.....	381
14.4	Splitting Application into Modules.....	383
14.4.1	Example of an Assembly File (Test1.asm).....	384
14.4.2	Corresponding include file (Test1.inc).....	384
14.4.3	Example of Assembly File (Test2.asm).....	385
14.4.3.1	Example of a PRM file (Test2.prm).....	385
14.5	Using Direct Addressing Mode to Access Symbols.....	386
14.5.1	Using the direct addressing mode to access external symbols.....	386
14.5.2	Using the direct addressing mode to access exported symbols.....	387
14.5.3	Defining symbols in the direct page.....	387
14.5.4	Using the force operator.....	388
14.5.5	Using SHORT sections.....	388

## Chapter 15 Appendices

15.1	Appendices.....	391
------	-----------------	-----

## Chapter 16 Global Configuration File Entries

16.1	[Installation] Section.....	393
16.1.1	Path.....	393
16.1.2	Group.....	394
16.2	[Options] Section.....	394
16.2.1	DefaultDir.....	394
16.3	[XXX_Assembler] Section.....	395
16.3.1	SaveOnExit.....	395
16.3.2	SaveAppearance.....	395
16.3.3	SaveEditor.....	395

Section number	Title	Page
16.3.4	SaveOptions.....	396
16.3.5	RecentProject0, RecentProject1.....	396
16.4	[Editor] Section.....	397
16.4.1	Editor_Name.....	397
16.4.2	Editor_Exe.....	397
16.4.3	Editor_Opts.....	398
16.5	Example.....	398

## Chapter 17 Local Configuration File Entries

17.1	[Editor] Section.....	401
17.1.1	Editor_Name.....	401
17.1.2	Editor_Exe.....	402
17.1.3	Editor_Opts.....	402
17.2	[XXX_Assembler] Section.....	403
17.2.1	RecentCommandLineX, X= integer.....	403
17.2.2	CurrentCommandLine.....	404
17.2.3	StatusbarEnabled.....	404
17.2.4	ToolbarEnabled.....	404
17.2.5	WindowPos.....	405
17.2.6	WindowFont.....	405
17.2.7	TipFilePos.....	406
17.2.8	ShowTipOfDay.....	406
17.2.9	Options.....	407
17.2.10	EditorType.....	407
17.2.11	EditorCommandLine.....	408
17.2.12	EditorDDEClientName.....	408
17.2.13	EditorDDETopicName.....	408
17.2.14	EditorDDEServiceName.....	409
17.3	Example.....	409

Section number	Title	Page
<b>Chapter 18</b>		
<b>MASM Compatibility</b>		
18.1	Comment Line.....	411
18.2	Constants (Integers).....	411
18.3	Operators.....	412
18.3.1	Directives.....	412
<b>Chapter 19</b>		
<b>MCUasm Compatibility</b>		
19.1	Labels.....	415
19.2	SET directive.....	415
19.3	Obsolete directives.....	416
<b>Chapter 20</b>		
<b>Assembler Messages</b>		
20.1	Assembler Messages.....	417
20.1.1	A1: Unknown message occurred .....	417
20.1.2	A2: Message overflow, skipping <kind> messages .....	418
20.1.3	A50: Input file '<file>' not found .....	418
20.1.4	A51: Cannot open statistic log file '<file>' .....	418
20.1.5	A52: Error in command line '<cmd>'.....	419
20.1.6	A53: Message <MessageId> is not used by this version. The mapping of this message is ignored. ....	419
20.1.7	A54: Option <Option> . .....	420
20.1.8	A56: Option value overridden for option <OptionName>. Old value '<OldValue>', new value '<NewValue>' .....	420
20.1.9	A64: Line Continuation occurred in <FileName> .....	420
20.1.10	A65: Environment macro expansion message '<description>' for <variablename> .....	421
20.1.11	A66: Search path <Name> does not exist .....	422
20.1.12	A1000: Conditional directive not closed .....	422
20.1.13	A1001: Conditional else not allowed here .....	424
20.1.14	A1002: CASE, DEFAULT or ENDSW detected outside from a SWITCH block .....	424
20.1.15	A1003: CASE or DEFAULT is missing .....	425

Section number	Title	Page
20.1.16	A1004: Macro nesting too deep. Possible recursion? Stop processing. (Set level with -MacroNest) .....	426
20.1.17	A1051: Zero Division in expression .....	427
20.1.18	A1052: Right parenthesis expected .....	428
20.1.19	A1053: Left parenthesis expected .....	429
20.1.20	A1054: References on non-absolute objects are not allowed when options -FA1 or -FA2 are enabled .....	429
20.1.21	A1055: Error in expression .....	430
20.1.22	A1056: Error at end of expression .....	431
20.1.23	A1057: Cutting constant because of overflow .....	432
20.1.24	A1058: Illegal floating point operation .....	432
20.1.25	A1059: != is taken as EQUAL .....	432
20.1.26	A1060: Implicit comment start .....	433
20.1.27	A1061: Floating Point format is not supported for this case .....	433
20.1.28	A1062: Floating Point number expected .....	433
20.1.29	A1101: Illegal label: label is reserved .....	434
20.1.30	A1103: Illegal redefinition of label .....	434
20.1.31	A1104: Undeclared user defined symbol: <Symbol> .....	435
20.1.32	A1105: Closing brace expected.....	436
20.1.33	A1106: Undeclared user defined symbol: <Symbol>. Symbol name is expected as macro argument.....	436
20.1.34	A1201: Label <Label> referenced in directive ABSENTRY is not absolute.....	437
20.1.35	A1202: ELF output: <details> .....	437
20.1.36	A1203: .....	438
20.1.37	A1251: Cannot open object file: Object file name too long .....	438
20.1.38	A1252: The exported label <name> is using an ELF extension .....	438
20.1.39	A1253: Limitation: code size > <SizeLimit> bytes .....	439
20.1.40	A1301: Structured type redefinition: <TypeName> .....	439
20.1.41	A1302: Type <TypeName> is previously defined as label .....	440
20.1.42	A1303: No type defined .....	441
20.1.43	A1304: Field <FieldName> is not declared in specified type .....	442
20.1.44	A1305: Type name expected .....	445

Section number	Title	Page
20.1.45	A1401: Value out of range -128..127 .....	448
20.1.46	A1402: Value out of range -32768..32767 .....	450
20.1.47	A1405: PAGE with initialized RAM not supported .....	450
20.1.48	A1406: HIGH with initialized RAM not supported .....	451
20.1.49	A1407: LOW with initialized RAM not supported .....	451
20.1.50	A1408: Out of memory, Code size too large .....	452
20.1.51	A1410: EQU or SET labels are not allowed in a PC relative addressing mode.....	452
20.1.52	A1411: PC Relative addressing mode is not supported to constants .....	453
20.1.53	A1412: Relocatable object '<Symbol>' not allowed if generating absolute file .....	454
20.1.54	A1413: Value out of relative range .....	455
20.1.55	A1414: Cannot set fixup.....	455
20.1.56	A1415: Cutting fixup overflow .....	456
20.1.57	A1416: Absolute section starting at <Address> size <Size> overlaps with absolute section starting at <Address> .....	456
20.1.58	A1417: Value out of possible range: Value<value> not in range [<LowValue>..<HighValue>].....	457
20.1.59	A1418: Negative offset is not propagated into the page in the fixup .....	458
20.1.60	A1419: Constant fixup failure <Description> .....	458
20.1.61	A1502: Reserved identifiers are not allowed as instruction or directive .....	458
20.1.62	A1503: Error in option -D: '<Description>' .....	459
20.1.63	A1601: Label must be terminated with a <:> .....	459
20.1.64	A1602: Invalid character at end of label (<LabelName>): semicolon or space expected .....	460
20.1.65	A1603: Directive, instruction or macro name expected: <Symbol> detected .....	460
20.1.66	A1604: Invalid character detected at the beginning of the line: <Character> .....	461
20.1.67	A1605: Invalid label name: <LabelName> .....	462
20.1.68	A2301: Label is missing .....	462
20.1.69	A2302: Macro name is missing .....	463
20.1.70	A2303: ENDM is illegal .....	464
20.1.71	A2304: Macro definition within definition .....	465
20.1.72	A2305: Illegal redefinition of instruction or directive name .....	466

Section number	Title	Page
20.1.73	A2306: Macro not closed at end of source .....	466
20.1.74	A2307: Macro redefinition .....	467
20.1.75	A2308: File name expected .....	468
20.1.76	A2309: File not found .....	469
20.1.77	A2310: Size specification expected .....	469
20.1.78	A2311: Symbol name expected .....	470
20.1.79	A2312: String expected .....	471
20.1.80	A2313: Nesting of include files exceeds 50 .....	472
20.1.81	A2314: Expression must be absolute .....	472
20.1.82	A2316: Section name required .....	473
20.1.83	A2317: Illegal redefinition of section name .....	474
20.1.84	A2318: Section not declared .....	474
20.1.85	A2319: No section link to this label .....	475
20.1.86	A2320: Value too small .....	475
20.1.87	A2321: Value too big .....	477
20.1.88	A2323: Label is ignored .....	477
20.1.89	A2324: Illegal Base (2,8,10,16) .....	478
20.1.90	A2325: Comma or Line end expected .....	479
20.1.91	A2326: Label <Label> is redefined .....	480
20.1.92	A2327: ON or OFF expected .....	481
20.1.93	A2328: Value is truncated .....	482
20.1.94	A2329: FAIL found .....	482
20.1.95	A2330: String is not allowed .....	483
20.1.96	A2332: FAIL found .....	484
20.1.97	A2333: Forward reference not allowed .....	485
20.1.98	A2335: Exported SET label<name> is not supported .....	485
20.1.99	A2336: Value too big .....	486
20.1.100	A2338: <FailReason> .....	487
20.1.101	A2340: Macro parameter already defined .....	488

Section number	Title	Page
20.1.102	A2341: Relocatable Section Not Allowed: an Absolute file is currently directly generated .....	488
20.1.103	A2342: Label in an OFFSET section cannot be exported .....	489
20.1.104	A2345: Embedded type definition not allowed .....	490
20.1.105	A2346: Directive or instruction not allowed in a type definition .....	491
20.1.106	A2350: MEXIT is illegal (detected outside of a macro) .....	492
20.1.107	A2351: Expected Comma to separate macro arguments .....	492
20.1.108	A2352: Invalid Character .....	493
20.1.109	A2353: Illegal or unsupported directive SECT .....	493
20.1.110	A2354: Ignoring directive '<directive>' .....	494
20.1.111	A2355: Illegal size specification .....	494
20.1.112	A2356: Illegal RAD50 character .....	494
20.1.113	A2357: Illegal macro argument '<Argument>' .....	495
20.1.114	A2358: Size prefix ignored .....	495
20.1.115	A2359: String index out of bounds.....	495
20.1.116	A2360: Maximum SUBSTR recursion level reached.....	496
20.1.117	A2380: Cutting very long line .....	496
20.1.118	A2381: Previous message was in this context '<Context>' .....	496
20.1.119	A2382: Illegal character ('\0') in source file .....	497
20.1.120	A2383: Input line too long .....	498
20.1.121	A2400: End of Line expected .....	499
20.1.122	A2401: Complex relocatable expression not supported .....	499
20.1.123	A2402: Comma expected .....	503
20.1.124	A2500: Equal expected .....	504
20.1.125	A2501: TO expected .....	505
20.1.126	A2502: ENDFOR missing .....	505
20.1.127	A2503: ENDFOR without FOR .....	506
20.1.128	A2600: Addition of values from a different address space .....	507
20.1.129	A2601: Subtraction of values from a different address space .....	507
20.1.130	A2602: Operation of values from a different address space .....	507

Section number	Title	Page
20.1.131	A3000: User requested stop .....	507
20.1.132	A4000: Recursive definition of label <Label name>=""> .....	508
20.1.133	A4001: Data directive contains no data .....	508
20.1.134	A4002: Variable access size differs from previous declaration .....	509
20.1.135	A4003: Found XREF, but no XDEF for label <Label>, ignoring XREF .....	509
20.1.136	A4004: Qualifier ignored .....	510
20.1.137	A4005: Access size mismatch for <Symbol> .....	510
20.1.138	A4006: Illegal value '<ErrorDescription>' .....	511
20.1.139	A4100: Address space clash for <Symbol> .....	511
20.1.140	A13001: Illegal Addressing Mode. <AddrModes> .....	512
20.1.141	A13003: Value is truncated to one byte .....	512
20.1.142	A13004: Value is truncated to two bytes .....	514
20.1.143	A13101: Illegal operand format .....	515
20.1.144	A13102: Operand not allowed .....	515
20.1.145	A13106: Illegal size specification .....	516
20.1.146	A13109: Positive value expected .....	517
20.1.147	A13110: Bit number expected .....	518
20.1.148	A13111: Value out of range .....	518
20.1.149	A13203: Not a HC08 instruction or directive .....	519
20.1.150	A13204: Instruction not supported by RS08 Core .....	520
20.1.151	A13205: RS08 instructions only supported in RS08 mode (use option -CrS08) .....	520
20.1.152	A13206: This instruction is only available for derivatives with MMU.....	521



# Chapter 1

## Using HC(S)08/RS08 Assembler

This document explains how to effectively use the HC(S)08/RS08 Macro Assembler.

### 1.1 Highlights

The major features of the HC(S)08/RS08 Assembler are:

- Graphical User Interface
- On-line Help
- 32-bit Application
- Conformation to the Freescale Assembly Language Input Standard

### 1.2 Structure of this Document

This section has the following chapters:

- [Working with Assembler](#) : Tutorial using the CodeWarrior Development Studio for Microcontrollers V10.x to create and configure an assembly-code project. In addition, there is a description of using the Assembler and the Linker as standalone Build Tools.
- [Assembler Graphical User Interface](#) : Description of the Macro Assembler's Graphical User Interface (GUI).
- [Environment](#) : Detailed description of the Environment variables used by the Macro Assembler.
- [Files](#) : Description of the input and output file the Assembler uses or generates.
- [Assembler Options](#) : Detailed description of the full set of assembler options.
- [Sections](#) : Description of the attributes and types of sections.

## Structure of this Document

- [Assembler Syntax](#) : Detailed description of the input syntax used in the assembly input files.
- [Assembler Directives](#) : List of every directive that the Assembler supports.
- [Macros](#) : Description of how to use macros with the Assembler.
- [Assembler Listing File](#) : Description of the assembler output files.
- [Mixed C and Assembler Applications](#) : Description of the important issues to be considered when mixing both the assembly and C source files in the same project.
- [Make Applications](#) : Description of special issues for the Linker.
- [How to...](#) : Examples of the assembly source code, linker PRM, and assembler output listings.
- [Assembler Messages](#) : Description of the assembler messages.

In addition to the chapters in this section, there are the following chapters of Appendices:

- [Global Configuration File Entries](#) : Description of the sections and entries that can appear in the global configuration file - `mcutools.ini`.
- [Local Configuration File Entries](#) : Description of the sections and entries that can appear in the local configuration file - `project.ini`.
- [MASM Compatibility](#) : Description of extensions for compatibility with the MASM Assembler.
- [MCUasm Compatibility](#) : Description of extensions for compatibility with the MCUasm Assembler.

## Chapter 2

# Working with Assembler

This chapter is primarily a tutorial for creating and managing HC(S)08/RS08 assembly projects with CodeWarrior Development Studio for Microcontrollers V10.x. In addition, there are directions to utilize the Assembler and Smart Linker Build Tools in the CodeWarrior Development Studio for assembling and linking assembly projects.

### NOTE

The CodeWarrior Development Studio tools actually support both the HC08 and HCS08 microcontroller derivatives. For brevity, this document uses the label *HC(S)08* to describe where the tools support both Microcontrollers derivatives. Where information is specific to the HC08, the label *HC08* is used, and where it is specific to the HCS08, the label *HCS08* is used.

In this chapter:

- [Programming Overview](#)
- [Managing Assembly Language Project Using CodeWarrior IDE](#)
- [Analysis of Groups and Files in a Project](#)
- [Writing your Assembly Source Files](#)
- [Analyzing Project Files](#)
- [Assembling Source Files](#)
- [Linking Application](#)
- [Directly Generating ABS File](#)
- [Assembler Build Properties Panels](#)

## 2.1 Programming Overview

In general terms, an embedded systems developer programs small but powerful microprocessors to perform specific tasks. These software programs for controlling the hardware are often referred to as firmware. One such use for firmware might be controlling small stepping motors in an automobile seat.

The developer instructs what the hardware should do with one or more programming languages, which have evolved over time. The three principal languages in use to program embedded microprocessors are C and its variants, various forms of C++, and assembly languages that are specially tailored to families of microcontrollers. C and C++ have been fairly standardized through years of use, whereas assembly languages vary widely and are usually designed by semiconductor manufacturers for specific families or even subfamilies, which are often called derivatives, of their embedded microprocessors.

Assembly language instructions are considered as being at a lower level (closer to the hardware) than the essentially standardized C statements. Programming in C may require some additional assembly instructions to be generated over and beyond what an experienced developer could do in straight assembly language to accomplish the same result. As a result, assembly language programs are usually faster to execute than C instructions, but require much more programming effort. In addition, each chip series usually has its own specialized assembly language which is only applicable for that family (or subfamily) of CPU derivatives.

Higher-level languages, such as C use assemblers to translate the syntax used by the programmer to the machine-language of the microprocessor, whereas assembly language uses assemblers. It is also possible to mix assembly and C source code in a single project. For more information, refer to the [Mixed C and Assembler Applications](#) chapter.

This manual covers the Assembler dedicated to the Freescale 8-bit HC(S)08/RS08 series of microcontrollers. There is a companion manual for this series that covers the HC(S)08 assembler.

The HC(S)08/RS08 Assembler can be used as a transparent, integral part of CodeWarrior Development Studio for Microcontrollers V10.x. This is the recommended way to get your project up and running in minimal time. Alternatively, the Assembler can also be configured and used as a standalone macro assembler as a member of Build Tool Utilities, such as a (Smart) Linker, Assembler, ROM Burner, Simulator, or Debugger.

The typical configuration of an Assembler is its association with a [Project Directory](#) and an [External Editor](#). The CodeWarrior software uses the project directory for storing the files it creates and coordinates the various tools integrated into the CodeWarrior suite. The Assembler is but one of these tools that the IDE coordinates for your projects. The tools used most frequently within the CodeWarrior IDE are its Editor, Compiler, Assembler, Linker, the Simulator/Debugger, and Processor Expert. Most of these *Build*

*Tools* are located in the `<MCU>\prog` subfolder of the CodeWarrior installation directory. The others are directly integrated into *CodeWarrior Development Studio for Microcontrollers V10.x*.

The textual statements and instructions of the assembly-language syntax are written by editors. The CodeWarrior IDE has its own editor, although any external text editor can be used for writing assembly code programs. If you have a favorite editor, chances are that it can be configured so as to provide both error and positive feedback from either the CodeWarrior IDE or the standalone Assembler.

## 2.1.1 Project Directory

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

There are three methods of designing a project.

- Start from scratch, make your project configuration (\*.ini) and layout files for use with the Build Tools,
- Use CodeWarrior IDE to coordinate and manage the entire project, or
- Begin project construction with CodeWarrior IDE and use the standalone build tools to complete the project.

### NOTE

The Build Tools (including Assembler, Compiler, Linker, Simulator/ Debugger, and others) are a part of the CodeWarrior Suite and are located in the `prog` folder in the CodeWarrior installation. The default location this folder is:

## 2.1.2 External Editor

The CodeWarrior IDE reduces programming effort because its internal editor is configured with the Assembler to enable error feedback. You can use the **Configuration** dialog box of the standalone Assembler or other standalone CodeWarrior Tools to configure or to select your choice of editors. Refer to the [Editor Setting Dialog Box](#) section of this manual.

## 2.2 Managing Assembly Language Project Using CodeWarrior IDE

The CodeWarrior IDE has an integrated wizard to easily configure and manage the creation of your project. The wizard will get your project up and running in short order by following a short series of steps to create and coordinate the project and to generate the basic files that are located in the project directory.

This section will create a basic CodeWarrior project that uses assembly source code. A sample program is included for a project created using the wizard. For example, the program included for an assembly project calculates the next number in a Fibonacci series. It is much easier to analyze any program if you already have some familiarity with solving the result in advance.

A Fibonacci series is an easily visualized infinite mathematical series:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... to infinity-->
```

It is simple to calculate the next number in this series. The first calculated result is actually the third number in the series because the first two numbers make up the starting point: 0 and 1. The next term in a Fibonacci series is the sum of the preceding two terms. The first sum is then:  $0 + 1 = 1$ . The second sum is  $1 + 1 = 2$ . The sixth sum is  $5 + 8 = 13$ . And so on to infinity.

Let's now create a project with the wizard and analyze the assembly source and the Linker's parameter files to calculate a Fibonacci series for a particular 8-bit microprocessor in the Freescale HC(S)08 family - **MC9S08GT60**. Along the way, some tips demonstrate how the CodeWarrior IDE helps manage your projects.

### 2.2.1 Create New Project

This section demonstrates creating a new project using the **New Bareboard Project** wizard.

1. Select **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > CodeWarrior**.

The **Workspace Launcher** dialog box appears, prompting you to select a workspace to use.

2. Click **OK** to accept the default workspace. To use a workspace different from the default, click the **Browse** button and specify the desired workspace.
3. Select **File > New > Bareboard Project** from the IDE menu bar.

The **New Bareboard Project** wizard launches - the **Create an MCU Bareboard Project** page appears.

4. Specify a name for the new project. For example, enter the project name as `Project_1`.
5. Click **Next**.

The **Devices** page appears.

6. Expand the tree control and select the derivative or board you would like to use. For example, select **S08 > HCS08G Family > MC9S08GT60**.
7. Click **Next**.

The **Connections** page appears.

8. Select the appropriate connection(s).
9. Click **Next**.

The **Languages** page appears.

10. Check the **Relocatable Assembly** checkbox and make sure that both the **C** and **C++** checkboxes are clear. By default **C** option is checked.
11. Click **Next**.

The **Rapid Application Development** page appears.

12. Select the appropriate rapid application development tool.
13. Click **Finish**.

The wizard creates a project according to your specifications. The newly created project is displayed in the **CodeWarrior Projects** view.

### NOTE

For detailed descriptions of the options available in the New Bareboard Project wizard pages, refer to the *Microcontrollers V10.x Targeting Manual*.

Select the project in the **CodeWarrior Projects** view. From the IDE menu bar, select **Project > Build Project** to build the project. The **Console** view displays the statements that direct the build tools to compile and link the project. The **Binaries** link appears, and so does the `FLASH` folder ([Additional Project Information](#)).

**NOTE**

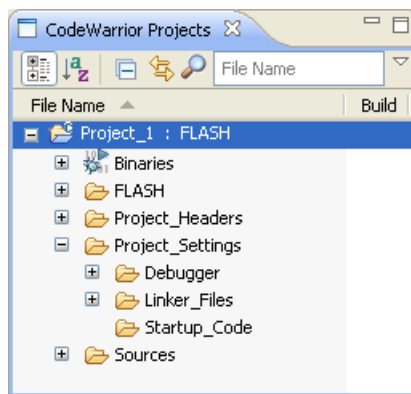
You can configure the IDE to build the project automatically. To configure the IDE to build the project automatically, check the **Build automatically** checkbox in the **Window > Preferences > General > Workspace** page.

## 2.2.2 Additional Project Information

The New Bareboard Project wizard sets up the HCS08 project in few minutes. You can add additional components to your project afterwards. A number of files and folders are automatically generated in the project folder. This folder is referred to in this manual as the project directory.

The major GUI component for your project is the **CodeWarrior Projects** view, as when the project is created, the project appears in the **CodeWarrior Projects** view in the Workbench window.

The following figure shows the project in the **CodeWarrior Projects** view.



**Figure 2-1. CodeWarrior Projects View**

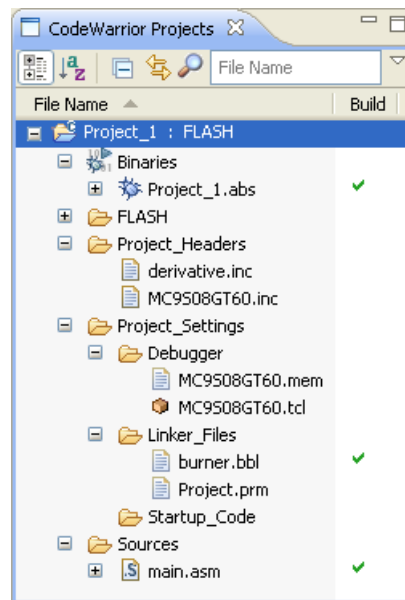
**NOTE**

The contents of the project directory vary depending upon the options selected while creating the project.

If you expand the folder icons, actually groups of files, by clicking in the **CodeWarrior Project** view, you can view the files created by the New Bareboard Project wizard.

The following figure shows the expanded project in the **CodeWarrior Projects** view.





**Figure 2-2. CodeWarrior Projects View Showing Files**

The expanded view displays the logical arrangement of the project files. At this stage, you can safely close the project and reopen it later, if desired.

The following is the list of default groups and files displayed in the project window.

- `Binaries` is a link to the generated binary (`.abs`) files.
- `FLASH` is the directory that contains all of the files used to build the application for `Project_1`. This includes the source, header, generated binary files, and the makefiles that manage the build process.
- `Project_Headers` is the directory that contains any Microcontrollers-specific header files.
- `Project_Settings` group contains the `Debugger` folder, the `Linker_Files` folder, and the `Startup_Code` folder.
  - The `Debugger` folder stores the memory configuration (`.mem`), launch configuration, and debug configuration file.
  - The `Linker_Files` folder stores the linker command file (`.prm`) and the burner command file (`.bbl`).
  - The `Startup_Code` folder has a C file that initializes the Microcontrollers's stack and critical registers when the program launches.
- The `Sources` folder contains the assembly source code files. For this example, the wizard has created the `main.asm` file.

Examine the project folder that the IDE generated when you created the project. To do this, right-click on the project's name (`Project_1 : FLASH`) in the **CodeWarrior Projects** view, and select **Show in Windows Explorer**. Windows displays the Eclipse workspace folder, along with the project folder, `Project_1`, within it.

These are the actual folders and files generated for your project. When working with standalone tools, you may need to specify the paths to these files, so it is best that you know their locations and functions.

Note that there are some files (`.project`, `.cproject`, and `.cwGenerateFileSetLog`) that store critical information about the project's state. The CodeWarrior Projects view does not display these files, and they should not be deleted.

The `FLASH\Sources` folder, which is created after the project is built, holds an object file for every assembly source-code file. In this case, `main.obj` is generated.

Double-click the `main.asm` file in the **Sources** group. The `main.asm` file opens in the editor area.

The following image displays the `main.asm` file in the editor view.

```

;*****
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more *
;* advanced functionality of this processor, please see the *
;* demonstration applications, located in the examples *
;* subdirectory of the "Freescale CodeWarrior for HC08" program *
;* directory. *
;*****

; Include derivative-specific definitions
    INCLUDE 'derivative.inc'

; export symbols
    XDEF _Startup, main
    ; we export both '_Startup' and 'main' as symbols. Either can
    ; be referenced in the linker .prm file or from C/C++ later on

    XREF __SEG_END_SSTACK ; symbol defined by the linker for the end of the stack

; variable/data section
MY_ZEROPAGE: SECTION SHORT ; Insert here your data definition

; code section
MyCode: SECTION
main:
_Startup:
    LDHX    #__SEG_END_SSTACK ; initialize the stack pointer
    TXS
    CLI      ; enable interrupts

mainLoop:
    ; Insert your code here
    NOP

    feed_watchdog
    BRA    mainLoop
    
```

**Figure 2-3. main.asm File**

You can use this sample `main.asm` file as a base to rewrite your own assembly source program. Otherwise, you can import other assembly-code files into the project and delete the default `main.asm` file from the project. For this project, the `main.asm` file contains the sample Fibonacci program.

## 2.3 Analysis of Groups and Files in a Project

In the **CodeWarrior Projects** view, the project files are distributed into four major groups, each with their own folder within the `Project_1` folder. You can add, rename, or delete files or groups, or you can move files or groups anywhere in the **CodeWarrior Projects** view.

### 2.3.1 CodeWarrior Groups

These groups and their usual functions are:

- Sources

This group contains the assembly source code files.

- Project\_Settings
  - Debugger

The `Debugger` folder stores the memory configuration file containing commands that define the legally accessible areas of memory for your specific part, the launch configuration file, and the debug configuration file.

- Linker Files

This group contains the burner file (`.bb1`), and the linker command file (`.prm`).

- Startup Code

This group contains the source code that manages the Microcontrollers initialization and startup functions. For HCS08 derivatives, these functions appear in the source file `start08.c`.

- Project\_Headers

This group holds include files. One include file is for the particular CPU derivative. In this case, the `MC9S08GT60.inc` file is for the MC9S08GT60 derivative.

#### NOTE

The default configuration of the project by the wizard does not generate an assembler output listing file for every `*.asm` source file. However, you can afterwards select **Generate a listing file** in the assembler options for the Assembler to generate a format-configurable listing file of the assembly source code (with the inclusion of include files, if desired). Assembler listing files (with the `*.lst` file extension) are

located in the project directory when \*.asm files are assembled with this option set.

This initial building of your project shows whether it is created and configured correctly. Now, you can utilize some of the CodeWarrior IDE features for managing your project.

However, it is not at all necessary to rename files and groups in the CodeWarrior IDE, so you can skip the following sections and resume the *Assembler* part of this tutorial at [Writing your Assembly Source Files](#).

## 2.3.2 Creating New Group

To create a new group:

1. Select **File > New > Other** from the IDE menu bar.

The **New** dialog box appears.

2. Expand the **General** tree node and select **Group**.
3. Click **Next**.

The **Group** wizard appears.

4. Enter the project directory to which you want to add the new group in the **Enter or select the parent folder** text box or select the required directory in the area below the **Enter or select the parent folder** text box.
5. Enter the name of the new group in the **Folder name** text box.
6. Click **Finish**.

The new *group* appears under the selected parent folder.

## 2.3.3 Adding New File to the Project

To add a new file to the project:

1. Select **File > New > Other** from the IDE menu bar.

The **New** dialog box appears.

2. Expand the **General** tree node and select **File**.
3. Click **Next**.

The **File** page appears.

4. Enter the project directory to which you want to add the new file in the **Enter or select the parent folder** text box or select the required directory in the area below the **Enter or select the parent folder** text box.
5. Enter the name of the new file with appropriate extension in the **File name** text box.
6. Click **Finish** .

The new *file* appears under the selected parent folder.

### 2.3.4 Renaming File or Group

To rename a file or group:

1. Right-click the file or group you want to rename in the **CodeWarrior Projects** view and select **Rename** from the context menu. Or, select the file or group and press **F2**. The **Rename Resource** dialog box appears.
2. Enter new name for the file or group and click **OK** or press **Enter**.

The selected file or group appears with the changes you made in the name.

### 2.3.5 Moving File

To move a file to a different group or project:

1. Right-click the file you want to move in the **CodeWarrior Projects** view and select **Move** from the context menu. The **Move Resources** dialog box appears.
2. Select the desired destination and click **OK**.

The file is moved to the selected location.

### 2.3.6 Removing File

To remove a file from a project:

1. Right-click the file in the **CodeWarrior Projects** view and select **Delete** from the context menu. The **Delete Resources** dialog box appears to confirm the deletion.
2. Click **Yes**.

This deletes the selected file from the project directory.

### 2.3.7 Restoring Deleted File

To restore a deleted file:

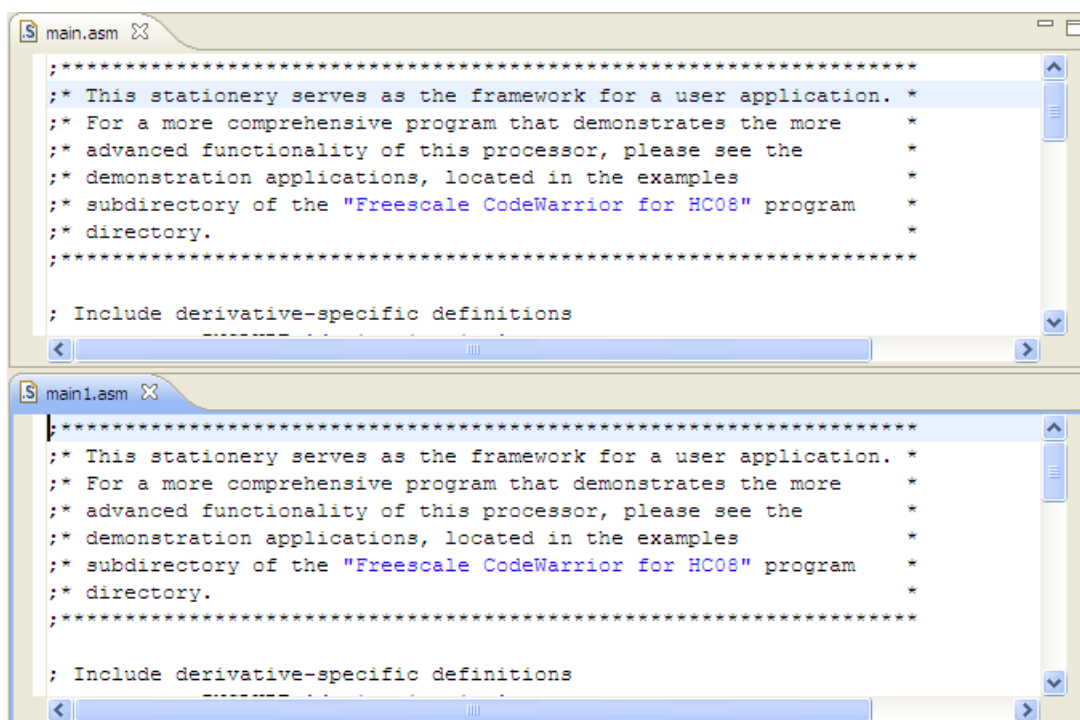
1. In the **CodeWarrior Projects** view, right-click the project to which the deleted file belongs.
2. Select **Restore from Local History** from the context menu. The **Restore from Local History** dialog box appears. The dialog box lists the deleted files available in the local history.
3. Check the required file checkbox and click **Restore**.

This restores the file to the original directory structure.

### 2.3.8 Using Editor

1. Double-click a file in the **CodeWarrior Projects** view to open the file in the editor area of the Workbench window.
2. With two or more files open in the editor area, select one of the editor tabs.
3. Holding down the left mouse button, drag the editor tab over the left, right, top, or bottom border of the editor area. Notice that the mouse pointer changes to a *drop cursor* that indicates where the editor tab will be moved when you release the mouse button. By dragging the editor tabs, you can tile the source files in the editor area in order to view source files side by side.

The following image shows the tiled source files in the editor area.



**Figure 2-4. Editor Area Showing Tiled Source Files**

4. Drag the borders of the editor area or each editor, to resize as desired.
5. Make desired changes in the source file. To save the file perform any of the following:
  - Select **File > Save** from the IDE menu bar.
  - Right-click the file and select **Save** from the context menu.
  - Press **CTRL+S**.
  - Click the **Save** icon on the toolbar.

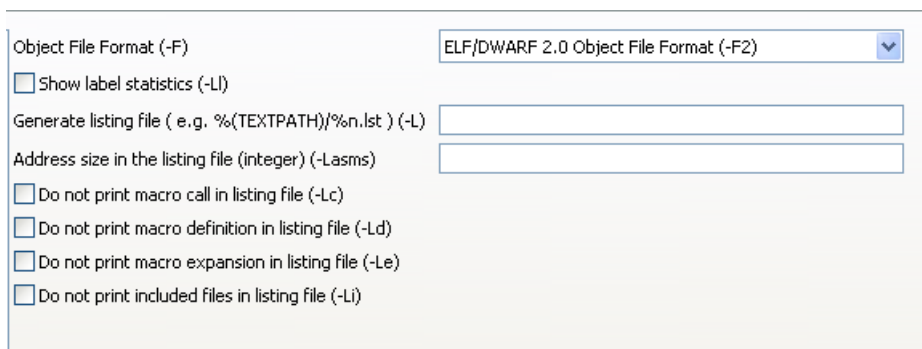
The file is saved with the changes you made using the editor.

### 2.3.9 Generating Listing Files

It was mentioned previously that the assembler output listing files are not generated without making configuration changes for the build target. To generate listing files, set up assembler options:

1. In the **CodeWarrior Projects** view, right-click the assembler project for which you want to generate output listing files.
2. Select **Properties** from the context menu that appears. The **Properties for <project name>** dialog box appears.
3. Select **C/C++ Build > Settings**. The **Tool Settings** page appears in the right panel.
4. Select **HCS08 Assembler > Output** in the **Tool Settings** page.



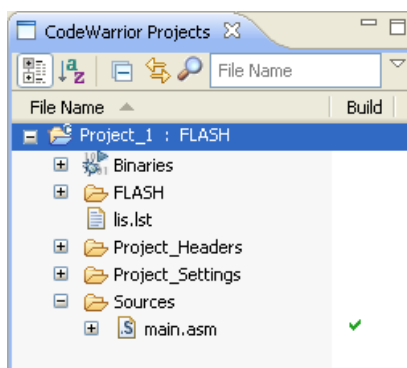


**Figure 2-5. HCS08 Assembler > Output Page**

5. Specifies the name, %n, and path, %TEXTPATH, of the assembly listing file in the **Generate listing file ( e.g. %(TEXTPATH)/%n.lst ) (-L)** text box. For example, enter `D:\Workspace\Project_1\lis.lst` in the text box, if `D:\Workspace\Project_1` and `lis.lst` are the path and name of the listing file.
6. Click **Apply** to save the modified settings.
7. Click **OK** to close the **Properties** dialog box.
8. Right-click the project in the **CodeWarrior Projects** view and select **Build Project** from the context menu.

The listing file appears in the **CodeWarrior Projects** view under the specified directory.

The following figure displays the assembly listing file in the **CodeWarrior Projects** view.



**Figure 2-6. CodeWarrior Projects View - Assembly Listing File**

## 2.4 Writing your Assembly Source Files

Once your project is configured, you can start writing your application's assembly source code and the Linker's PRM file.

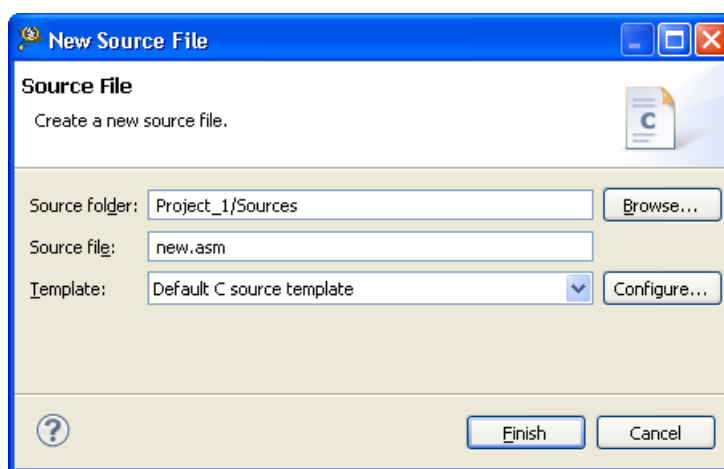
## NOTE

You can write an assembly application using one or several assembly units. Each assembly unit performs one particular task. An assembly unit is comprised of an assembly source file and, perhaps, some additional include files. Variables are exported from or imported to the different assembly units so that a variable defined in an assembly unit can be used in another assembly unit. You create the application by linking all of the assembly units.

The usual procedure for writing an assembly source-code file is to use the editor that is integrated into the CodeWarrior IDE.

To create a new assembly source file:

1. Select **File > New > Source File** from the IDE menu bar. The **New Source File** dialog box appears.



**Figure 2-7. New Source File Dialog Box**

2. Enter the folder in which you want to add the new file in the **Source Folder** text box or click **Browse** to select the desired folder, for example `Project_1/Sources`.
3. Enter the name of the new file with extension `*.asm` in the **Source File** text box.
4. Click **Finish**.

A newly created file opens in the editor area. Write your assembly source code in the file.

## 2.5 Analyzing Project Files

We will analyze the default `main.asm` file that was generated when the project was created with the New Bareboard Project wizard. The following listing shows the assembler source code for the Fibonacci program.

### Listing: main.asm file

```

;*****
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more *
;* advanced functionality of this processor, please see the *
;* demonstration applications, located in the examples *
;* subdirectory of the "Freescale CodeWarrior for HC08" program *
;* directory. *
;*****
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'

; export symbols
        XDEF _Startup, main
        ; we export both '_Startup' and 'main' as symbols. Either
can
        ; be referenced in the linker .prm file or from C/C++ later
on
        XREF __SEG_END_SSTACK ; symbol defined by the linker for
the end of the stack
; variable/data section
MY_ZEROPAGE: SECTION SHORT ; Insert here your data definition
; code section
MyCode: SECTION
main:
_Startup:
        LDHX #__SEG_END_SSTACK ; initialize the stack pointer
        TXS
        CLI; enable interrupts
mainLoop:
        ; Insert your code here
        NOP
        feed_watchdog
        BRA mainLoop
    
```

Since the RS08 memory map is different from the HC08 memory map (and so is the instruction set), The following listing shows a similar example for RS08.

### NOTE

In order to assemble files for the RS08 derivative, pass the `-Crs08` option to the assembler. To pass the `-Crs08` option to the assembler, click the **Code Generation** tab in the **HC08 Assembler Option Settings** dialog box. Check the **Derivative Family** checkbox. From the option buttons that are displayed, select **RS08 Derivative Family**.

### Listing: Contents of Source File for RS08 Derivative

```

;*****
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more *
;* advanced functionality of this processor, please see the *
;* demonstration applications, located in the examples *
;* subdirectory of the "Freescale CodeWarrior for HC08" program *
;* directory. *
;*****
; export symbols
        XDEF _Startup, main
        ; we export both '_Startup' and 'main' as symbols. Either
can
        ; be referenced in the linker .prm file or from C/C++ later
on
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
;$$IF CLI
        ; enable interrupts
;$$// we should include here MCUInit.inc. Unfortunately, the one that
Unis generates does not assemble -> fix this when the fixed it.
;$$//; Include device initialization code
;$$//
        INCLUDE 'MCUInit.inc'
;
        XREF MCU_init
;$$ENDIF
; variable/data section
TINY_RAM_VARS: SECTION  RS08_SHORT
definition
Counter:    DS.B    1
FiboRes:   DS.B    1

```

```

tmpCounter: DS.B 1
tmp: DS.B 1
; code section
MyCode: SECTION
main:
  _Startup:
  ;$$IF CLI ; enable
  interrupts

  ; Call generated Device Initialization function

  ; JSR MCU_init
  ;$$ENDIF
mainLoop:
  CLRA ; A contains counter
cntLoop:  INCA
  CBEQA #14,mainLoop ; larger values cause overflow.

  MOV #HIGH_6_13(SRS),PAGESEL
  STA MAP_ADDR_6(SRS) ; feed the watchdog
  STA Counter ; update global.
  BSR CalcFibo
  STA FiboRes ; store result
  LDA Counter
  BRA cntLoop ; next round.
CalcFibo: ; Function to calculate fibonacci numbers. Argument is in A.
  DBNZA fiboDo ; fiboDo
  INCA
  RTS
fiboDo:
  STA tmpCounter ; the counter
  CLRX ; second last = 0
  LDA #$01 ; last = 1
FiboLoop: STA tmp ; store last
  ADDX
  LDX tmp
  DBNZ tmpCounter,FiboLoop
    
```

```
FiboDone:
```

```
        RTS                ; result in A
```

When writing your assembly source code, pay special attention to the following:

- Make sure that symbols outside the current source file (in another source file or in the linker configuration file) that are referenced from the current source file are externally visible. Notice that we have inserted the assembly directive `XDEF_Startup, main` where appropriate in the example.
- In order to make debugging from the application easier, we strongly recommend that you define separate sections for code, constant data (defined with `DC`) and variables (defined with `DS`). This will mean that the symbols located in the variable or constant data sections can be displayed in the data window component.
- Make sure to initialize the stack pointer when using the `BSR` or `JSR` instructions in your application. The stack can be initialized in the assembly source code and allocated to RAM memory in the Linker parameter file, if a `*.prm` file is used.

### NOTE

The default assembly project created using the New Bareboard Project wizard initializes the stack pointer automatically with a symbol defined by the Linker for the end of the stack `__SEG_END_SSTACK`. For the RS08 derivative, initializing the stack does not apply.

## 2.6 Assembling Source Files

Once an assembly source file is available, you can assemble it. Either use the CodeWarrior IDE to assemble the `*.asm` files or use the standalone assembler of the build tools in the `<MCU>\prog` folder in the CodeWarrior installation.

### 2.6.1 Assembling and Linking with CodeWarrior IDE

The CodeWarrior IDE simplifies the assembly of your assembly source code. To assemble and link all the files in the project, select the project in the **CodeWarrior Projects** view and select **Project > Build Project** from the IDE menu bar. The files generated after assembling and linking the project are placed into the `<CPU Derivative>` subfolder in the project directory. The files include:

- `< assembly_source_file >.dbg`

This file contains symbolic debugging information.

- `<project_name>.abs`

This is the final executable file.

- `<project_name>.map`

This Linker map file lists the names, load addresses, and lengths of all segments in your program. In addition, it lists the names and load addresses of any groups in the program, the start address, and messages about any errors the Linker encounters.

Also, when you build a project, the project's source code files assemble into object (\*.obj) files.

The object files are generated and placed into the `<CPU Derivative>\Sources` subfolder in the project directory. The path of the object file created on assembling the `main.asm` file is:

`<project directory>\<CPU Derivative>\Sources\main.obj`

The Wizard does not generate default assembler-output listing files. If you want such listing files generated, follow the steps in topic [Generating Listing Files](#).

You can add the \*.lst files to the project window for easier viewing. This way you do not have to continually hunt for them with your editor.

## 2.6.2 Assembling with Assembler

It is also possible to use the HC(S)08/RS08 Assembler as a standalone assembler. If you prefer not to use the assembler but you want to use the Linker, you can skip this section and proceed to [Linking Application](#).

This tutorial does not create another project from scratch with the Build Tools, but instead uses some files of a project already created by the New Bareboard Project wizard. The CodeWarrior IDE can create, configure, and manage a project much easier and quicker than using the Build Tools. However, the Build Tools could also create and configure another project from scratch.

A Build Tool, such as the Assembler makes use of a project directory file for configuring and locating its input and generated files. The folder that is designated for this purpose is referred to by a Build Tool as the *current directory*.

Start the Assembler by double-clicking the `ahc08.exe` file in the `<MCU>\prog` folder in the CodeWarrior installation directory. The *Assembler* opens (refer to the figure listed below). Read the tip displayed in the **Tip of the Day** dialog box, if you want to, and then click **Close** to close the dialog box.

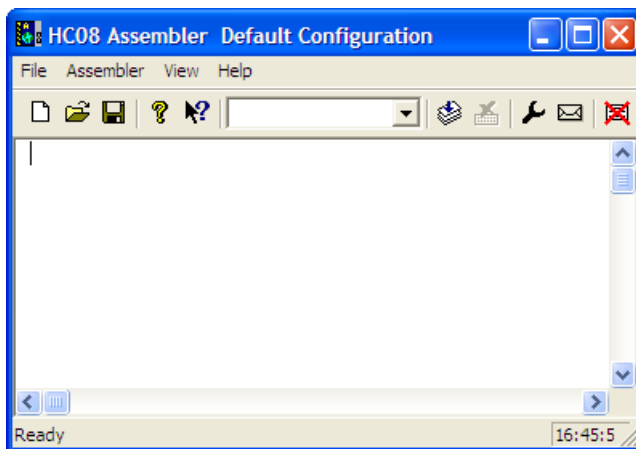


Figure 2-8. HC08 Assembler Default Configuration Dialog Box

### 2.6.2.1 Configuring Assembler

A Build Tool, such as the Assembler, requires information from the 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 listing the most recent projects, etc. All tools may store some global data into the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (for example, `C:\WINDOWS`), as the following listing shows.

**Listing: Typical locations for a global configuration file**

```
\<CWInstallDir>\MCU\prog\mcutools.ini - #1 priority
C:\WINDOWS\mcutools.ini - used if there is no mcutools.ini file above
```

If a tool is started in the default location `<CWInstallDir>\CW MCU V10.x\MCU\prog` directory, the initialization file in the same directory as the tool is used:

```
<CWInstallDir>\MCU\prog\mcutools.ini
```



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

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

- Local

This file could be used by any Build Tool for a particular project. For information about entries for the local configuration file, refer to the section [Local Configuration File Entries](#) in the Appendices.

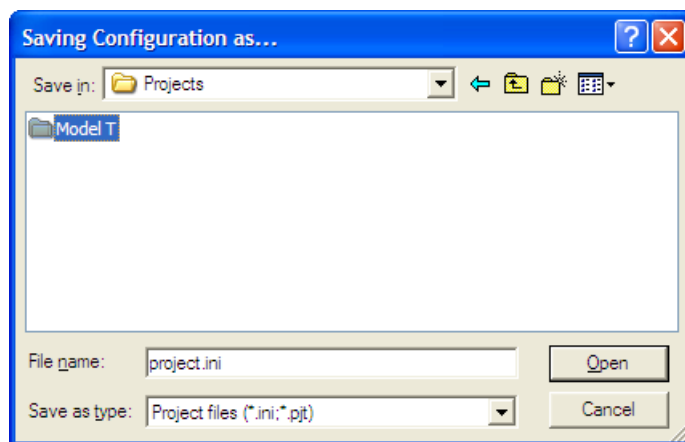
After opening the Assembler, you would load the configuration file for your project if it already had one. However, you will create a new configuration file for the project in this tutorial and save it so that when the project is reopened, its previously saved configuration state is used.

Now let's save this configuration in a newly created folder that will become the project directory.

1. Select **File > New / Default Configuration** to open a new default configuration.
2. Select **File > Save Configuration As** to save this configuration.

The **Saving Configuration as** dialog box appears.

3. Navigate to the desired location and click the **Create New Folder** icon on the dialog box toolbar.
4. Enter a name for the project directory.



**Figure 2-9. Loading Configuration Dialog Box**

5. Click **Open**.

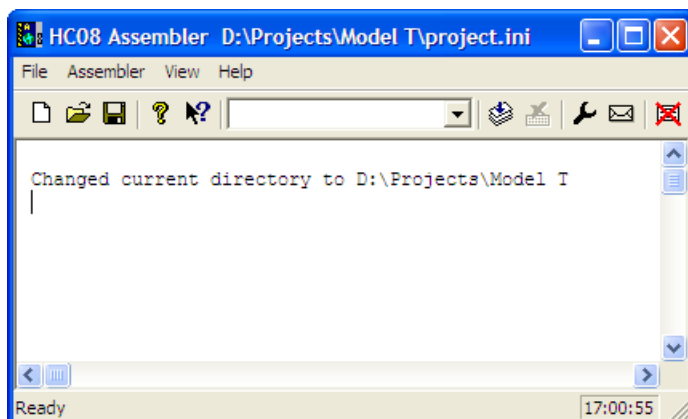
In this case, `ModelT` becomes the project directory in the `Projects` folder.

6. Click **Save**.

The `project.ini` file is created in the `ModelT` folder and becomes the local configuration file for this project.

The current directory for the Microcontroller Assembler is changed to your project directory.

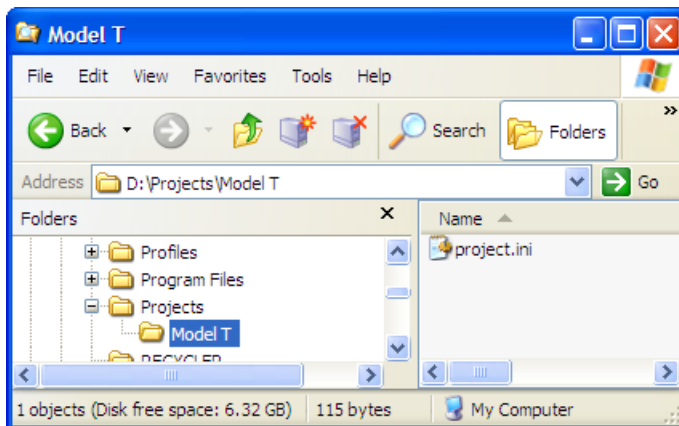
The following image shows the assembler window displaying the current directory.



**Figure 2-10. Assembler Displaying Current Directory**

If you were to examine the project directory with the Windows Explorer at this point, it would only contain the `project.ini` configuration file that the Assembler just created.

The following image displays the `project.ini` configuration file that the Assembler just created.



**Figure 2-11. Project directory in Windows Explorer**

If you further examined the contents of the `project.ini` configuration file, you would see that it contains Assembler options in the `[AHC08_Assembler]` portion of the file. The `project.ini` file for this project only has an `[AHC08_Assembler]` section, as the following listing displays.

## Listing: Contents of project.ini file

```
[AHC08_Assembler]
StatusBarEnabled=1

ToolBarEnabled=1

WindowPos=0,1,-1,-1,-1,-1,680,151,1148,491

EditorType=4
```

The `AHC08_Assembler` options are described in detail in [\[XXX\\_Assembler\] Section](#) in the Appendices.

Next, you have to set the object-file format that you will use (HIWARE or ELF/DWARF).

1. Select **Assembler > Options**.

The **HC08 Assembler Option Settings** dialog box (refer to the figure listed below) appears.

2. Click the **Output** tab. Check the **Generate a listing file** checkbox.
3. Check the **Object File Format** checkbox. Select **ELF/DWARF 2.0 Object File Format** from the drop-down list box displayed in the **Output** page for the **Object File Format** checkbox.
4. Check the **Do not print included files in list file** checkbox if you want the listing file to be shorter.

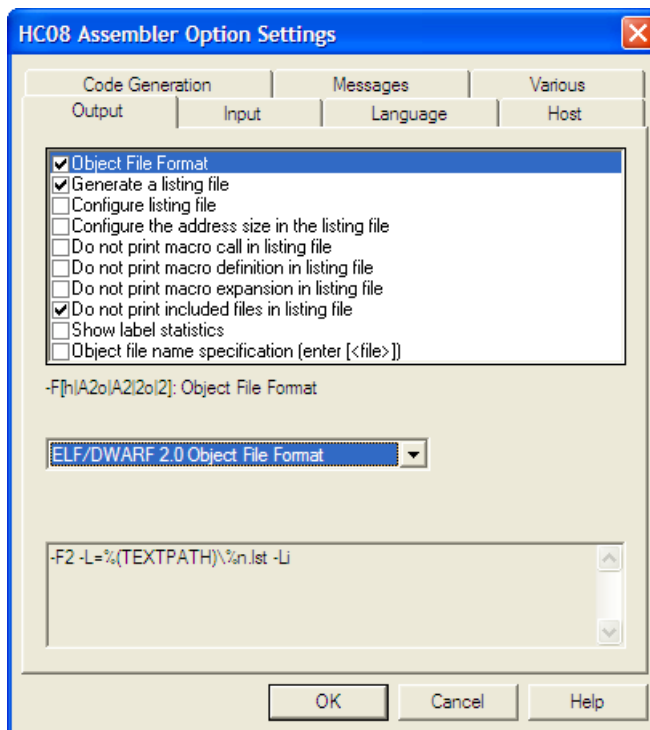


Figure 2-12. HC08 Assembler Option Settings Dialog Box

5. Click **OK** to close the **HC08 Assembler Option Settings** dialog box.

Save the changes to the configuration by:

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

The assembler is now set with the object-file format that you have selected.

**NOTE**

For the RS08 derivative the HIWARE Object File Format is not supported.

The following listing shows the `project.ini` file's contents, after the changes to the configuration are saved.

**Listing: project.ini file with additional assembly options**

```
[AHC08_Assembler]
StatusBarEnabled=1

ToolbarEnabled=1

WindowPos=0,1,-1,-1,-1,-1,680,151,1148,491

EditorType=4

Options=-F2 -L=%(TEXTSPATH)\%n.lst -Li
```

**2.6.2.2 Input Files**

Now that the project's configuration is set, you can assemble an assembly-code file. However, the project does not contain any source-code files at this point. You could create assembly `*.asm` and include `*.inc` files from scratch for this project. However, for simplicity's sake, you can copy and paste the `main.asm` and the `derivative.inc` files from the previous CodeWarrior project.

For this project, you should have a project directory named `Model T`. Within this folder, you should have another folder named `Sources`, which contains the two files described above. Using a text editor of your choice, modify the `main.asm` file so that it appears as the following listing shows:

**Listing: main.asm File**

```
*****
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more *
;* advanced functionality of this processor, please see the *
;* demonstration applications, located in the examples *
;* subdirectory of the "CodeWarrior for Microcontrollers V6.1" *
```

```

;* program directory. *
;*****
; export symbols
XDEF _Startup, main
; we use export '_Startup' as symbol. This allows us to
; reference '_Startup' either in the linker .prm file
; or from C/C++ later on
XREF __SEG_END_SSTACK ; symbol defined by the linker
; for the end of the stack
; Include derivative-specific definitions
INCLUDE 'derivative.inc'
; variable/data section
MY_ZEROPAGE: SECTION SHORT ; Insert here your data definition
Counter: DS.B 1
FiboRes: DS.B 1
; code section
MyCode: SECTION
main:
_Startup:
_LDHX #__SEG_END_SSTACK ; initialize the stack pointer
TXS
CLI ; enable interrupts
mainLoop:
CLRA ; A contains counter
cntLoop: INCA
CBEQA #14,mainLoop ; larger values cause overflow.
STA Counter ; update global.
BSR CalcFibo
STA FiboRes ; store result
LDA Counter
BRA cntLoop ; next round.
; Function to calculate fibonacci numbers. Argument is in A.
CalcFibo:
DBNZA fiboDo ; fiboDo
INCA
RTS
fiboDo:
PSHA ; the counter
CLRXL ; second last = 0
LDA #01 ; last = 1
FiboLoop: PSHA
ADD 1,SP
PULX
DBNZ 1,SP,FiboLoop
FiboDone: PULH ; release counter
RTS ; result in A
    
```

Now there are three files in the project:

- the `project.ini` configuration file and
- two files in the `Sources` folder:
  - `main.asm`
  - `derivative.inc`

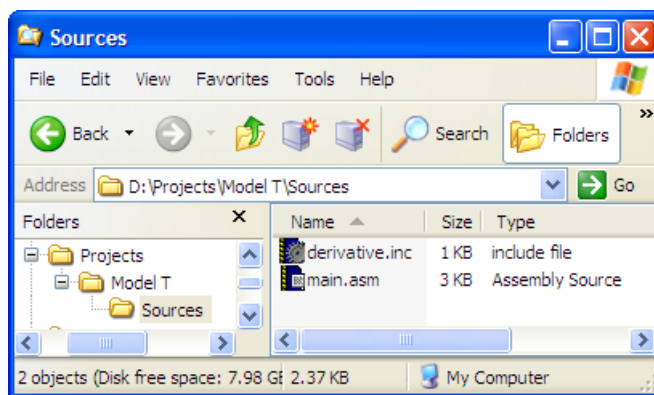


Figure 2-13. Project Files

### 2.6.2.3 Assembling Assembly Source-code Files

Let's assemble the `main.asm` file.

1. Select **File > Assemble** from the menu bar.
2. The **Select File to Assemble** dialog box appears. Browse to the `Sources` folder in the project directory and select the `main.asm` file.

The following image shows `main.asm` file.

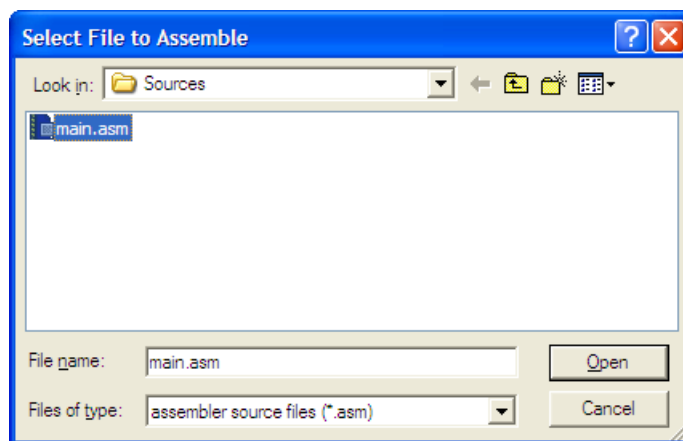
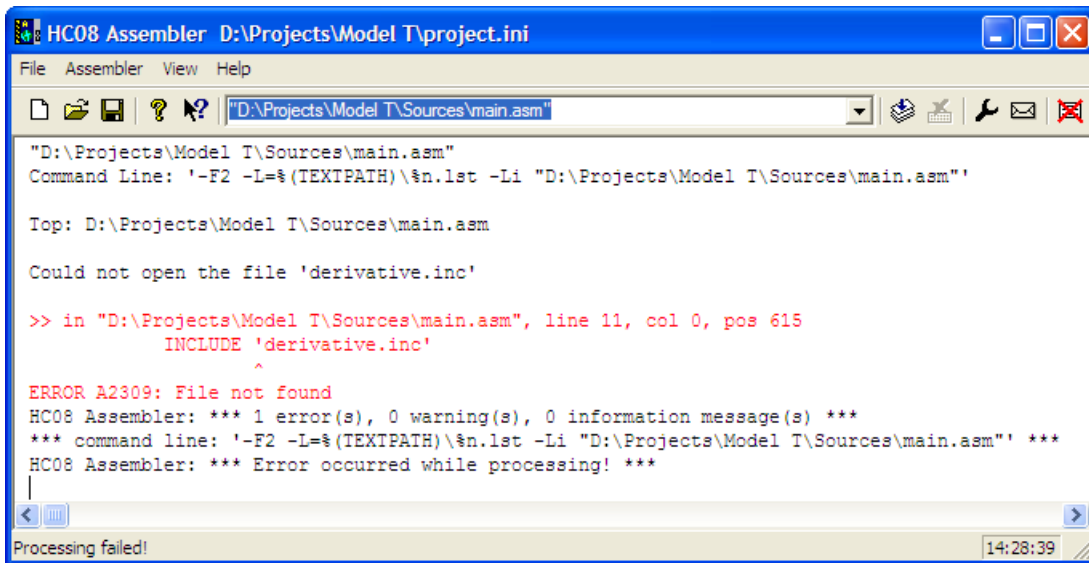


Figure 2-14. Select File to Assemble Dialog Box

3. Click **Open**.

The `main.asm` file starts assembling.

The following image shows the result of assembling of `main.asm` file.

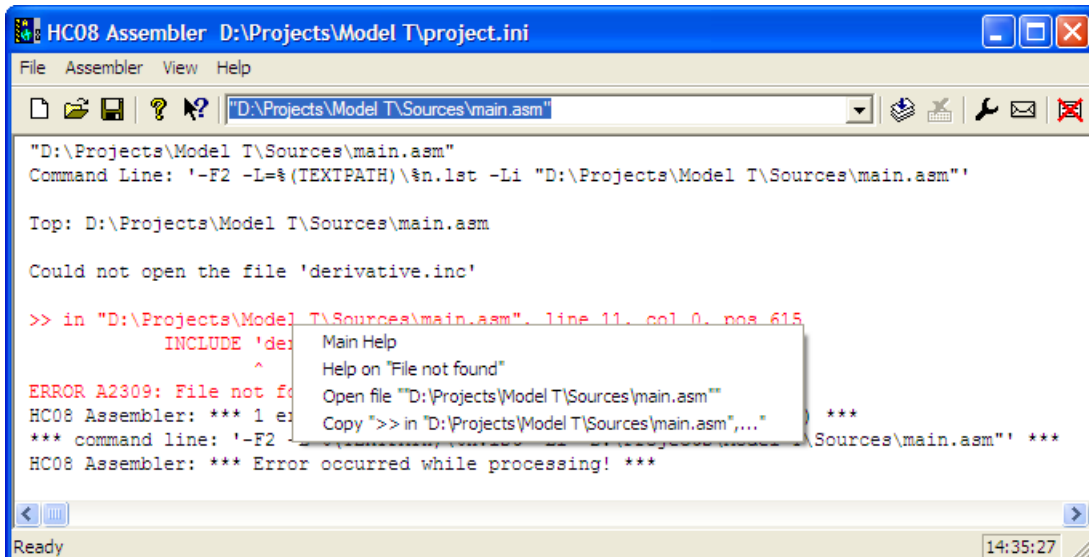


**Figure 2-15. Results of Assembling main.asm File**

The project window provides information about the assembly process or generates error messages if the assembly was unsuccessful. In this case, the *A2309 File not found* error message is generated. As the following image displays, if you right-click on the text containing the error message, a context menu appears.

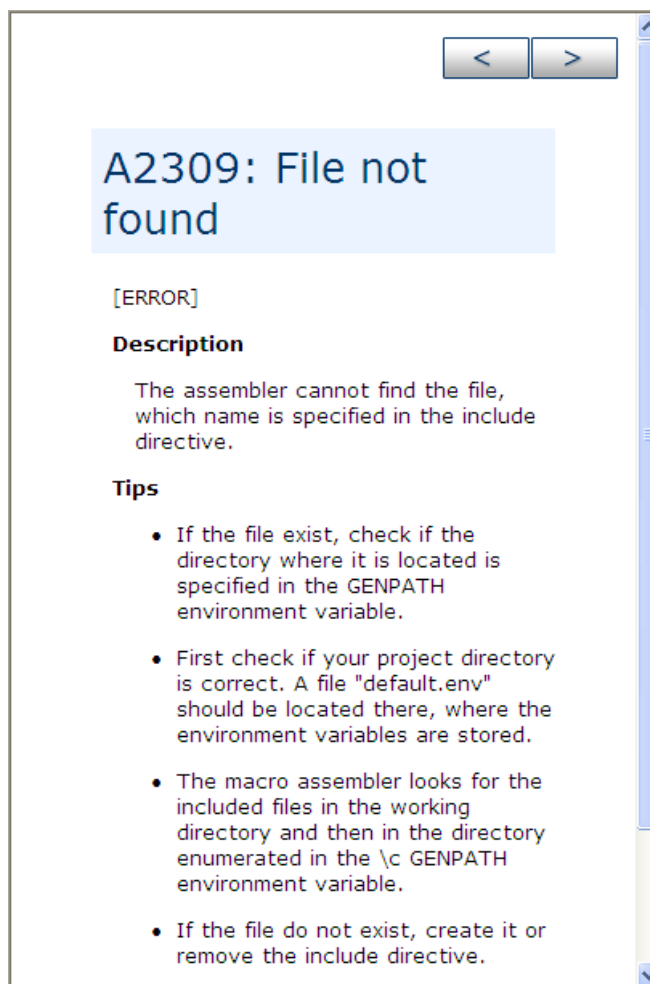
### NOTE

If you get any other types of errors, make sure the `main.asm` file is modified as shown in [Listing: main.asm File](#).



**Figure 2-16. Context Menu**

Select **Help on "file not found"** and help for the *A2309* error message appears, as the following image displays.



**Figure 2-17. A2309: File not found**

You know that the file exists because it is included in the `Sources` folder that you imported into the project directory. The help message for the `A2309` error states that the Assembler is looking for this "missing" include file first in the current directory and then in the directory specified by the `GENPATH` environment variable. This suggests that the `GENPATH` environment variable should specify the location of the `derivative.inc` include file.

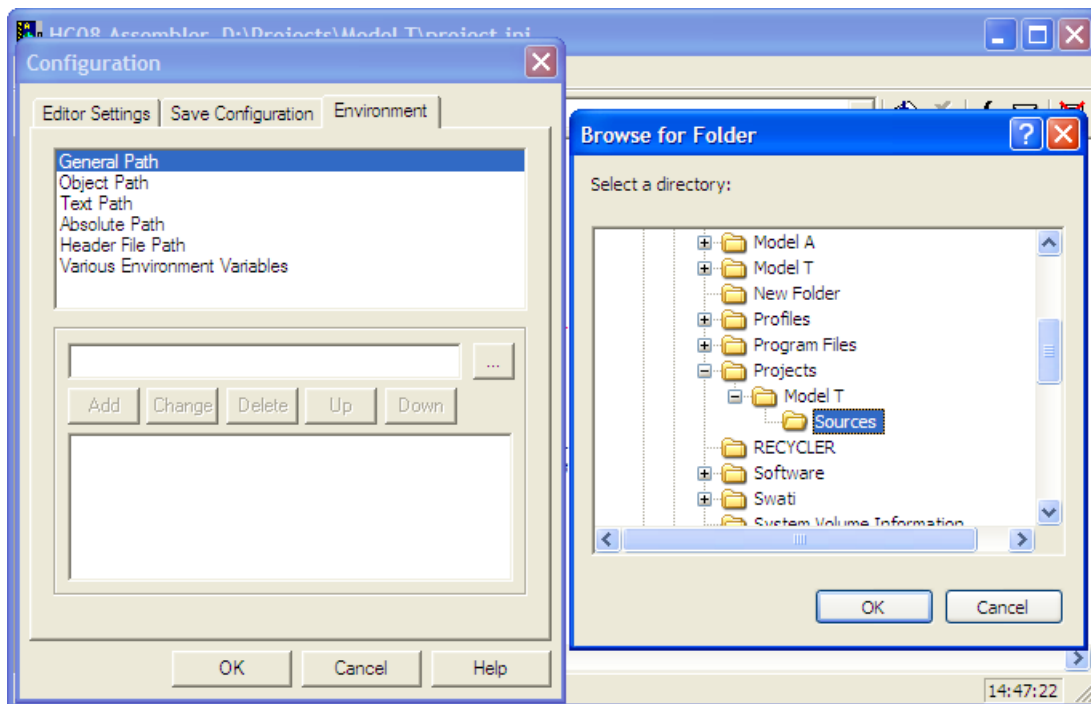
#### NOTE

If you read the `main.asm` file, you could have anticipated this on account of this statement on line 20: `INCLUDE 'derivative.inc'`.

To fix this error:

1. Select **File > Configuration**.
2. The **Configuration** dialog box appears (refer to the figure listed below). Click the **Environment** tab and then select **General Path**.
3. Press the "..." button and navigate in the **Browse for Folder** dialog box for the folder that contains the `derivative.inc` file - the `Sources` folder in the project directory.

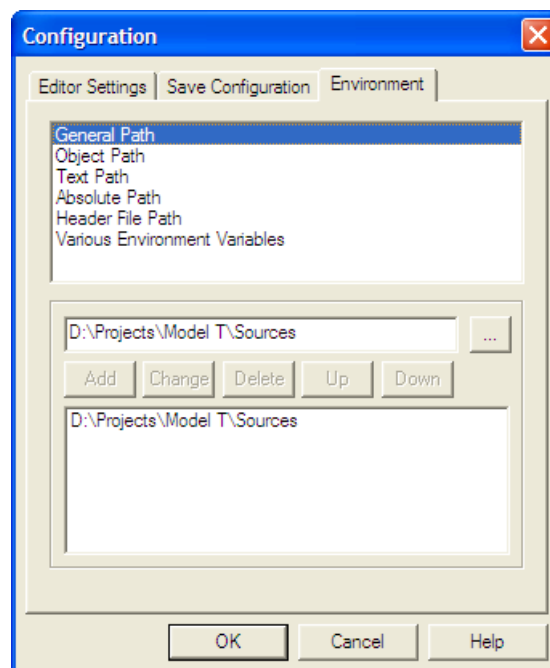




**Figure 2-18. Browsing for Sources Folder**

4. Click **OK** to close the **Browse for Folder** dialog box.
5. The **Configuration** dialog box is active again (refer to the figure listed below). Click the **Add** button.

The path to the `derivative.inc` file " `D:\Projects\Model T\Sources`" appears in the area below the **Add** button.



**Figure 2-19. Adding GENPATH**

**6. Click OK.**

An asterisk appears in the title bar of the Assembler window, so save the change to the configuration.

**7. Click the Save button in the toolbar or select File > Save Configuration.**

The asterisk disappears.

The new path is updated in the `derivative.inc` file.

### Tip

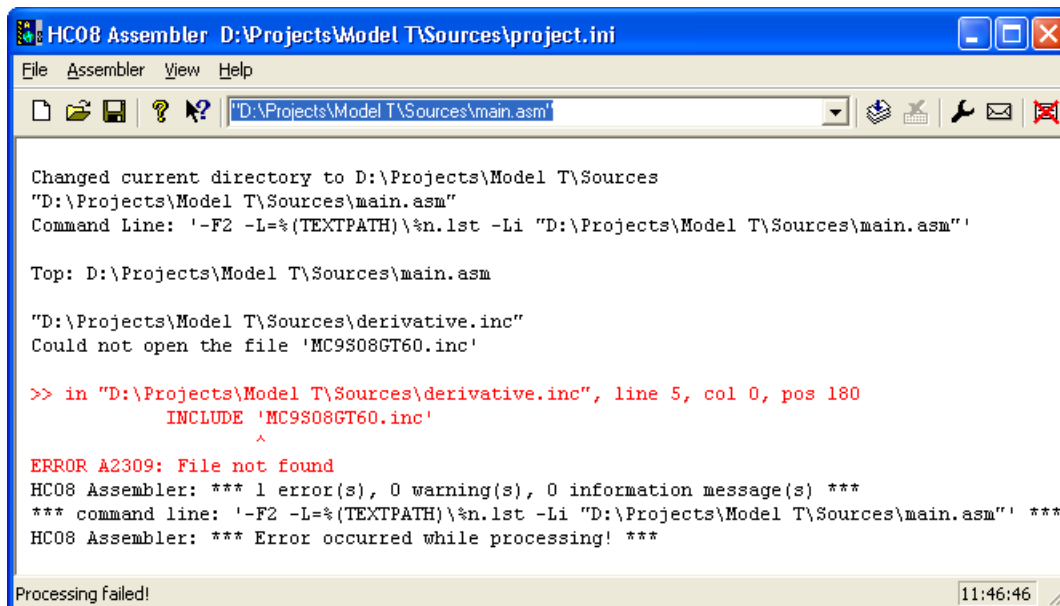
You can clear the messages in the Assembler window at any time by selecting **View > Log > Clear Log**.

Now that you have supplied the path to the `derivative.inc` file, assemble the `main.asm` file again.

Select **File > Assemble** and again navigate to the `main.asm` file and click **Open**. However, the `A2309` error message reappears but this time for a different include file - `mc9s08ac128.inc`.

### NOTE

In this case, the `derivative.inc` file has this statement: `INCLUDE 'mc9s08ac128.inc'`. Therefore, a prior reading of the assembly-code and include files suggests these include files might require `GENPATH` configurations. If possible, set any needed `GENPATH` in advance of assembling the source-code files.



**Figure 2-20. Assemble Attempt #2**

Fix this by repeating the `GENPATH` routine for the other include file. The `mc9s08ac128.inc` file is located at this path:

```
<CWInstallDir>\MCU\lib\hc08c\device\asm_include
```

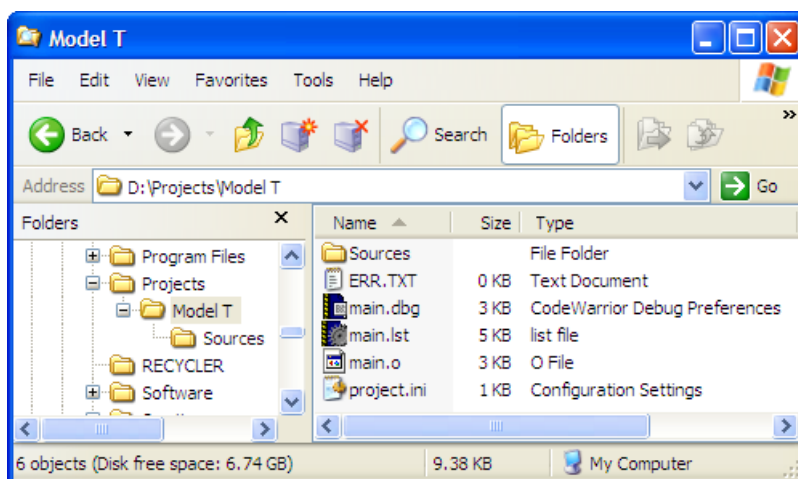
`CWInstallDir` is the directory in which the CodeWarrior software is installed.

The `asm_include` folder is the typical place for missing include files.

After the `GENPATH` is set up for the second include file and saved as before, you can try to assemble the `main.asm` file for the third time.

The Macro Assembler indicates successful assembling and indicated that the Code Size was 39 bytes. The message `*** 0 error(s)`, indicates that the `main.asm` file assembled without errors. Do not forget to save the configuration one additional time.

The Assembler also generated a `main.dbg` file (for use with the Simulator/Debugger), a `main.o` object file (for further processing with the Linker), and a `main.lst` output listing file in the project directory. The binary object-code file has the same name as the input module, but with the `*.o` extension - `main.o`. The debug file has the same name as the input module, but with the `*.dbg` extension - `main.dbg` and the assembly output listing file has the `*.lst` extension. The following image displays the project directory after the successful assembly of the project.



**Figure 2-21. Project Directory After Successful Assembly**

The `ERR.TXT` file is present in the project directory because of the earlier failed attempts at assembling. The `ERR.TXT` file is empty after a successful assembly. You can delete this file. The following listing shows the `project.ini` file.

### Listing: `project.ini` file after `GENPATH` environment variable is created

```
[AHC08_Assembler]
StatusBarEnabled=1

ToolBarEnabled=1
```

## Linking Application

```

WindowPos=0,1,-32000,-32000,-1,-1,290,513,903,833

EditorType=4

Options=-F2 -L=%(TEXTPATH)\%n.lst -Li

CurrentCommandLine="D:\Projects\Model T\Sources\main.asm"

RecentCommandLine0="D:\Projects\Model T\Sources\main.asm"

RecentCommandLine1=D:\Workspace\test\Sources\main.asm

[Environment Variables]

GENPATH=C:\Freescale\CW MCU
v10.3\MCU\lib\hc08c\device\asm_include;D:\Projects\Model T\Sources

OBJPATH=

TEXTPATH=

ABSPATH=

LIBPATH=

```

The haphazard running of this project was intentionally designed to fail to illustrate what occurs if the path of any `include` file is not properly configured. Be aware that `include` files may be included by either `*.asm` or `*.inc` files. In addition, remember that the `lib` folder in the CodeWarrior installation contains several derivative-specific `include` and `prm` files available for inclusion into your projects.

## 2.7 Linking Application

Once the object files are available you can link your application. The linker organizes the code and data sections into ROM and RAM memory areas according to the project's linker parameter (PRM) file.

### 2.7.1 Linking with CodeWarrior IDE

The Linker's input files are object-code files from the assembler and compiler, the library files, and the Linker PRM file.

#### 2.7.1.1 PRM File

If you are using the CodeWarrior IDE to manage your project, a pre-configured PRM file for a particular derivative is already set up, as the following listing displays.

### Listing: Linker PRM file for mc9s08gt60 derivative - Project.prm

```

/* This is a linker parameter file for the mc9s08gt60 */
NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */

    Z_RAM                =  READ_WRITE    0x0080 TO 0x00FF;

    RAM                  =  READ_WRITE    0x0100 TO 0x107F;

    ROM                  =  READ_ONLY     0x182C TO 0xFFAF;

    ROM1                 =  READ_ONLY     0x1080 TO 0x17FF;

    ROM2                 =  READ_ONLY     0xFFC0 TO 0xFFCB;

/* INTVECTS
for Interrupt Vectors */
    =  READ_ONLY         0xFFCC TO 0xFFFF; Reserved

END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */

    DEFAULT_RAM,                /* non-zero page variables */
                                INTO  RAM;

    _PRESTART,                  /* startup code */

    STARTUP,                    /* startup data structures */

    ROM_VAR,                    /* constant variables */

    STRINGS,                    /* string literals */

    VIRTUAL_TABLE_SEGMENT,      /* C++ virtual table segment */

    DEFAULT_ROM,

    COPY                        /* copy down information: how
to initialize variables */
                                INTO  ROM; /* ,ROM1,ROM2: To use
"ROM1,ROM2" as well, pass the option -OnB=b to the compiler */

    _DATA_ZEROPAGE,            /* zero page variables */

    MY_ZEROPAGE                 INTO  Z_RAM;

END

STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for
an application. */
    
```

The following listing is an example Linker PRM file for the RS08 derivative.

### Listing: Linker PRM file for RS08 derivative - Project.prm

## Linking Application

```
NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */
SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */
```

```
TINY_RAM           =  READ_WRITE   0x0005 TO 0x000D;
RAM                =  READ_WRITE   0x0020 TO 0x004F;
RESERVED_RAM       =  NO_INIT      0x0000 TO 0x0004;
ROM                =  READ_ONLY    0x3C00 TO 0x3FF7;
```

```
END
```

```
PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */
```

```
RESERVED           INTO RESERVED_RAM;
TINY_RAM_VARS      INTO TINY_RAM;
DIRECT_RAM_VARS    INTO RAM, TINY_RAM;
DEFAULT_RAM        INTO RAM, TINY_RAM;
DEFAULT_ROM        INTO ROM;
```

```
END
```

```
STACKSIZE 0x00 /* no stack for RS08 */
```

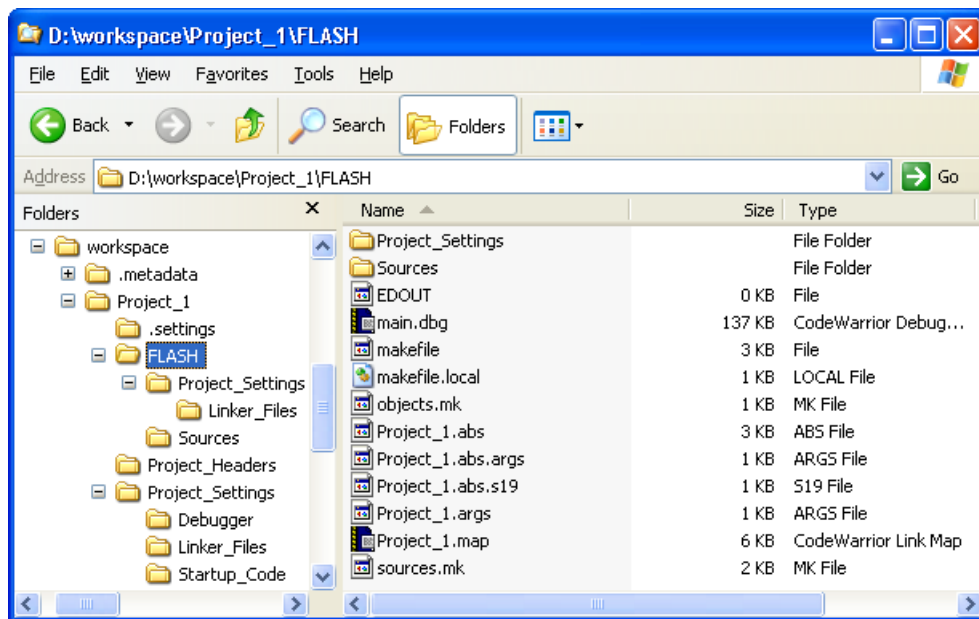
```
VECTOR 0 _Startup /* Reset vector: this is the default entry point for
an application. */
```

The Linker PRM file allocates memory for the stack and the sections named in the assembly source code files. If the sections in the source code are not specifically referenced in the `PLACEMENT` section, then these sections are included in `DEFAULT_ROM` OR `DEFAULT_RAM`.

The `STACKSIZE` entry is used to set the stack size. The size of the stack for this project is 80 bytes. Some entries in the Linker PRM file may be commented-out by the IDE, as are the three last items in the `Project.prm` file in the listing, [Linker PRM file for mc9s08gt60 derivative - Project.prm](#).

### 2.7.1.2 Linking Object-code Files

You can build this relocatable assembly project by selecting **Project > Build Project** from the IDE menu bar. When the project is built, the Linker generates a `*.abs` and a `*.map` file in the `FLASH` subfolder in the project directory, as the following image displays.



**Figure 2-22. Contents of <CPU Derivative> Subfolder After Building Project**

The `Project_1.abs` and `Project_1.map` files in the figure above are the Linker output files resulting from the object-code and PRM files.

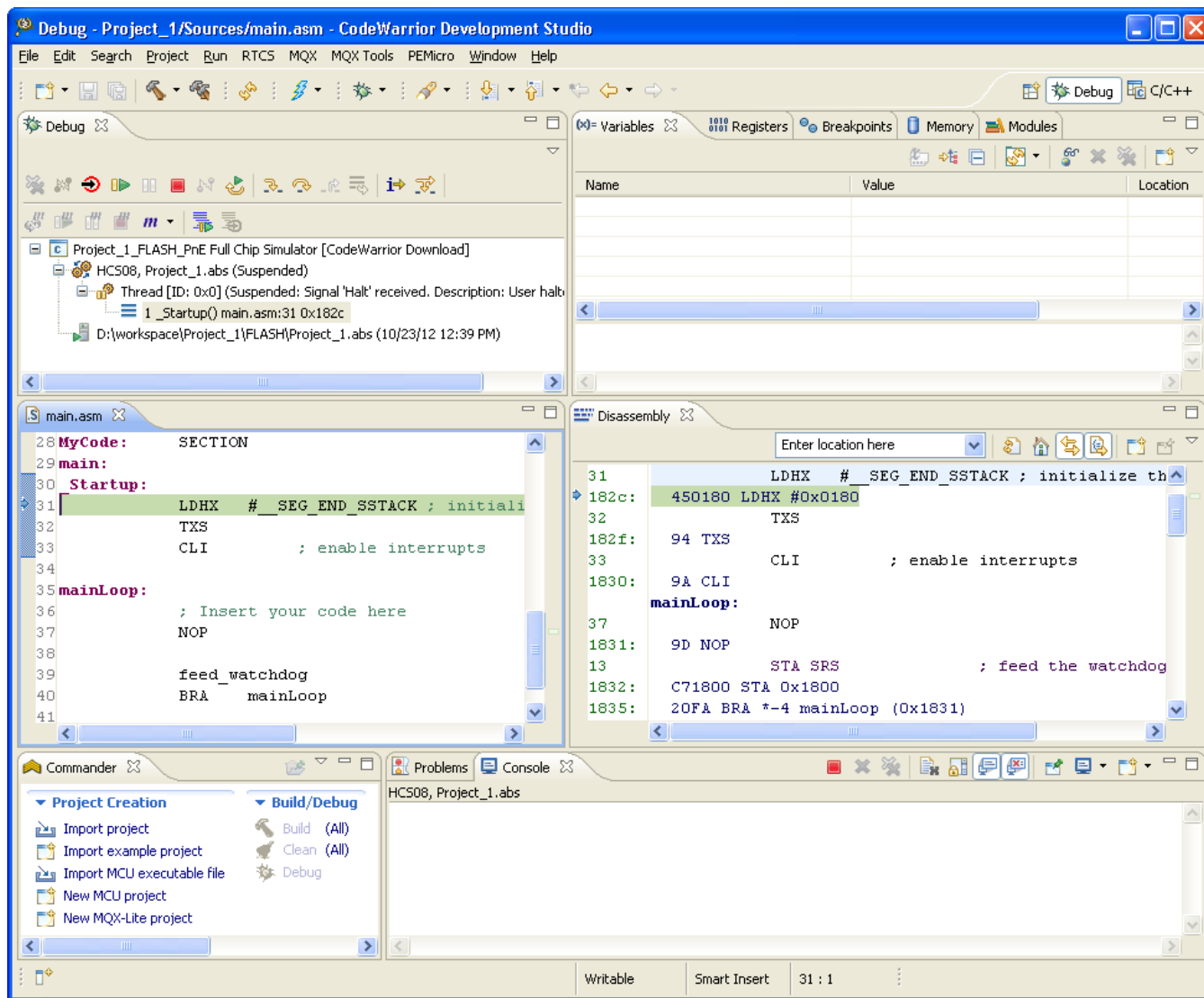
To debug the project:

1. From the main menu bar of the IDE, select **Run > Debug Configurations** .

The **Debug Configurations** dialog box appears. The left side of this dialog box has a list of debug configurations that apply to the current application.

2. Expand the **CodeWarrior Download** configuration.
3. From the expanded list, select the debug configuration that you want to modify.
4. Click the **Debugger** tab. The **Debugger** page appears in the area beneath the tabs.
5. Change the settings on this page as per your requirements. For example, select the required target processor and simulator/emulator.
6. Click the **Apply** button to save the new settings.
7. Click the **Debug** button to start the debugging session.

The perspective switches to the **Debug** perspective displaying the debugging process, as the following image displays.



**Figure 2-23. Debug Perspective**

In the Debug perspective, you can control your program's execution by setting breakpoints, suspending launched programs, stepping through your code, and examining the values of variables.

The Debug perspective displays information about:

- The stack frame of the suspended threads of each target that you are debugging
- Each thread in your program represented as a node in the tree
- The process of each program that you are running

The Debug perspective also drives the Source view. As you step through your program, the Source view highlights the location of the execution pointer.



## 2.7.2 Linking with Linker

If you are using the Linker (*SmartLinker*) build tool utility for a relocatable assembly project, you will use a PRM file for the Linker to allocate ROM and RAM memory areas.

1. Using a text editor, create the project's linker parameter file. You can modify a \*.prm file from another project and rename it as <project\_name>.prm.
2. Store the PRM file in a convenient location, such as the project directory.
3. In the <project\_name>.prmfile, change the name of the executable (\*.abs) file to whatever you choose, for example, <project\_name>.abs. In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's Model T.prm file (a PRM file for MC9S08GT60 from another CodeWarrior project was adapted), as the following listing shows.

### Listing: Layout of a PRM file for the Linker - Model T.prm

```

/* This is a linker parameter file for the mc9s08gt60 */
LINK Model_T.abs /* Absolute executable file */

NAMES main.o /* Input object-code files are listed here. */

END

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */

    Z_RAM                =  READ_WRITE    0x0080 TO 0x00FF;
    RAM                  =  READ_WRITE    0x0100 TO 0x107F;
    ROM                  =  READ_ONLY     0x182C TO 0xFFAF;
    ROM1                 =  READ_ONLY     0x1080 TO 0x17FF;
    ROM2                 =  READ_ONLY     0xFFC0 TO 0xFFCB;
    /* INTVECTS          =  READ_ONLY     0xFFCC TO 0xFFFF; Reserved
for Interrupt Vectors */

END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */

    DEFAULT_RAM,                /* non-zero page variables */
                                INTO  RAM;

    _PRESTART,                  /* startup code */
    STARTUP,                    /* startup data structures */
    ROM_VAR,                    /* constant variables */
    STRINGS,                    /* string literals */
    VIRTUAL_TABLE_SEGMENT,      /* C++ virtual table segment */
    DEFAULT_ROM,

```

## Linking Application

```

COPY                                     /* copy down information: how
to initialize variables */

                                     INTO ROM; /* ,ROM1,ROM2: To use
"ROM1,ROM2" as well, pass the option -OnB=b to the compiler */

_DATA_ZEROPAGE,                          /* zero page variables */
MY_ZEROPAGE                               INTO Z_RAM;

END

STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for
an application. */

```

### NOTE

If you are adapting a PRM file from a CodeWarrior project, all you really need to add is the `LINK` portion and the object-code filenames to be linked in the `NAMES` portion.

The default size for the stack using the New Bareboard Project wizard for `MC9S08GT60` is 80 bytes - (`STACKSIZE 0x50`). This Linker statement and `__SEG_END_SSTACK` in the assembly-code snippet below determine the size and placement of the stack in RAM:

```

MyCode: SECTION ; code section

main:

_Startup:

LDHX #__SEG_END_SSTACK ; initialize stack pointer

TXS

```

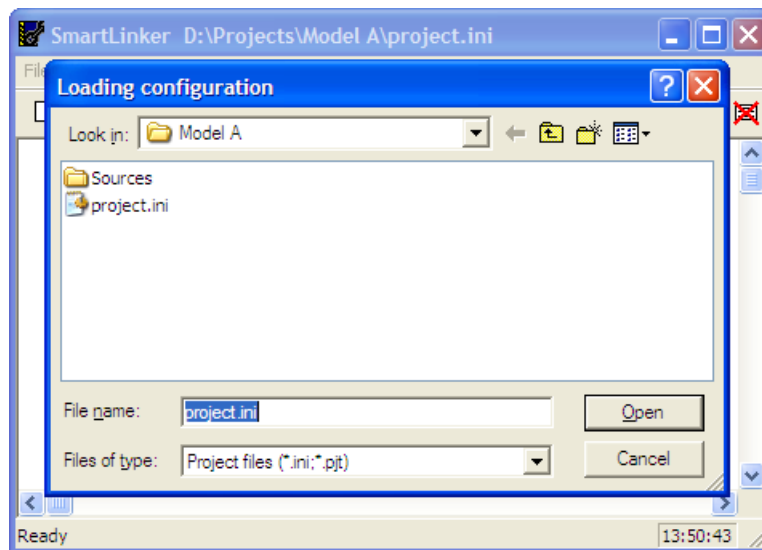
The statements in the linker parameter file are described in the Linker portion of the Build Tool Utilities manual.

4. Start the SmartLinker tool by double-clicking `linker.exe` located in the `<MCU>\prog` folder in the CodeWarrior installation directory.
5. Click **Close** to close the **Tip of the Day** dialog box.
6. Load the project's configuration file.

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

7. Select **File > Load Configuration** and navigate to and select the project's configuration file.

The following image displays the **Loading configuration** dialog box.

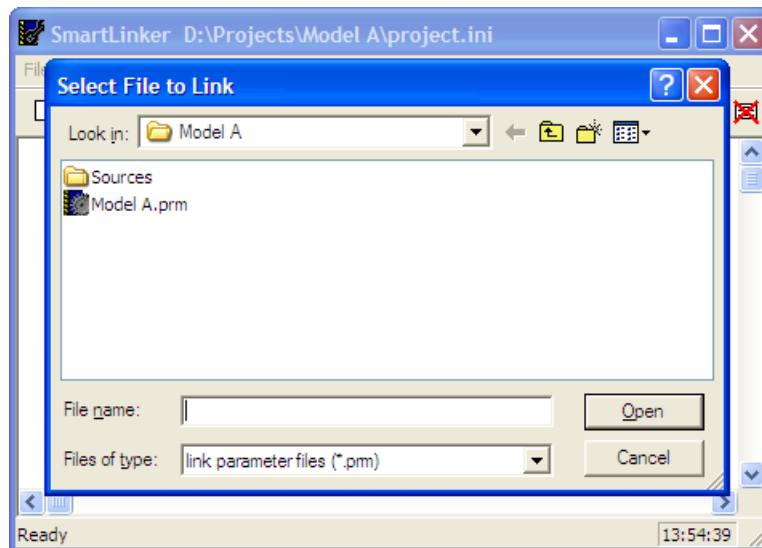


**Figure 2-24. Microcontroller Linker**

8. Click **Open** to load the configuration file.

The project directory is now the current directory for the Linker.

9. Select **File > Save Configuration** to save the configuration.
10. Select **File > Link**. The **Select File to Link** dialog box appears (refer to the image listed below).
11. Browse to locate and select the PRM file for your project.



**Figure 2-25. Select File to Link Dialog Box**

12. Click **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 following image displays the smart linker window after linking.

```

SmartLinker D:\Projects\Model A\project.ini
File SmartLinker View Help
"D:\Projects\Model A\Model A.prm"
"D:\Projects\Model A\Model A.prm"
Command Line: "D:\Projects\Model A\Model A.prm"
Reading Parameters
Linking D:\Projects\Model A\Model A.prm
Read Binary Input Files
Reading file 'D:\Projects\Model A\main.o'
Marking Referenced Objects
Moving Objects across Sections
Reserving Memory for Startup Data
Allocating Objects
Preparing Startup Data
Generating Code
Generating Symbol table
Generating DWARF data version 2.0
Code Size: 13
Generating MAP file 'D:\Projects\Model A\Model_A.map'
SmartLinker: *** 0 error(s), 0 warning(s), 0 information message(s) ***
SmartLinker: *** Processing ok ***
Processing ok 13:56:31

```

**Figure 2-26. SmartLinker Window After Linking**

The messages in the linker's project window indicate that:

- The current directory for the Linker is the project directory, `D:\Projects\Model A`.
- The `Model A.prm` file is used to name the executable file, which object files are linked, and how the RAM and ROM memory areas are allocated for the relocatable sections. The Reset and application entry points are also specified in this file.
- There is one object file, `main.o`.
- The output format is DWARF 2.0.
- The Code Size is 13 bytes.
- A Linker Map file, `Model_A.map` is generated.
- No errors or warnings occur and no information messages are issued.

The `TEXTPATH` environmental variable was not used for this project. Therefore, the Linker generates its `*.map` Linker Map file in the same folder that contains the PRM file for the project. Because the `ABSPATH` environment variable was not used, the `*.abs` executable file is generated in the same folder as the Linker PRM file. The following image shows the contents of the project directory after the relocatable assembly project is linked.

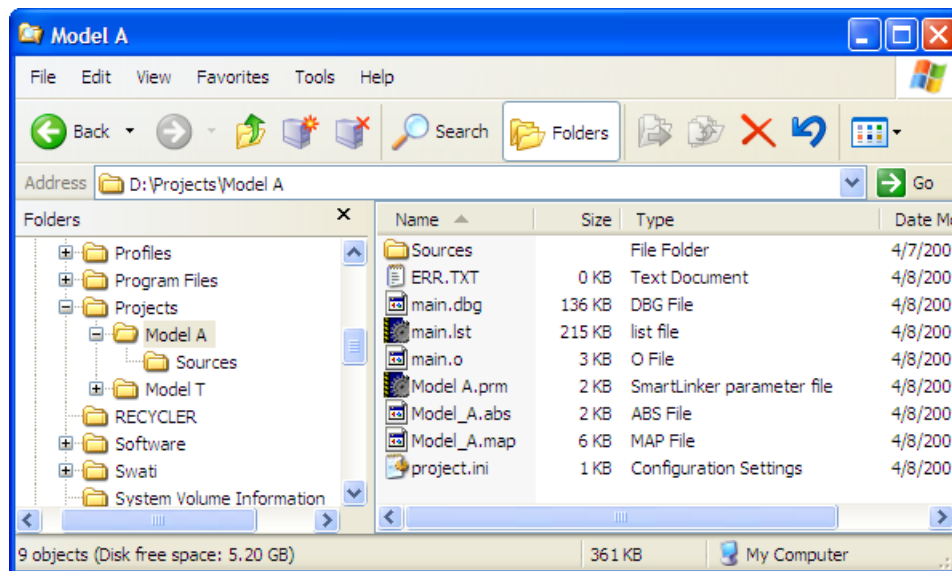


Figure 2-27. Project Directory After Linking

## 2.8 Directly Generating ABS File

You can use the Assembler build tool or CodeWarrior IDE to generate an `ABS` file directly from your assembly-source file. The Assembler may also be configured to generate an S-Record File at the same time.

When you use the Assembler or IDE to directly generate an `ABS` file, there is no Linker involved. This means that the application must be implemented in a single assembly unit and must contain only absolute sections.

### 2.8.1 Creating Absolute Assembly Project

To directly generate an `ABS` file, you need to create an absolute assembly project:

1. Start the CodeWarrior for Microcontrollers V10.x IDE.
2. Select **File > New > Bareboard Project** from the IDE menu bar.

The **New Bareboard Project** wizard launches - the **Create an MCU Bareboard Project** page appears.

3. Specify a name for the new project. For example, enter the project name as `AbsoluteAssembly`.
4. Click **Next**.

The **Devices** page appears.

- Expand the tree control and select the derivative or board you would like to use.
- Click **Next**.

The **Connections** page appears.

- Select the appropriate connection(s).
- Click **Next**.

The **Languages** page appears.

- Clear the **C** checkbox, which is checked by default, to enable the **Absolute Assembly** checkbox.
- Check the **Absolute Assembly** checkbox.

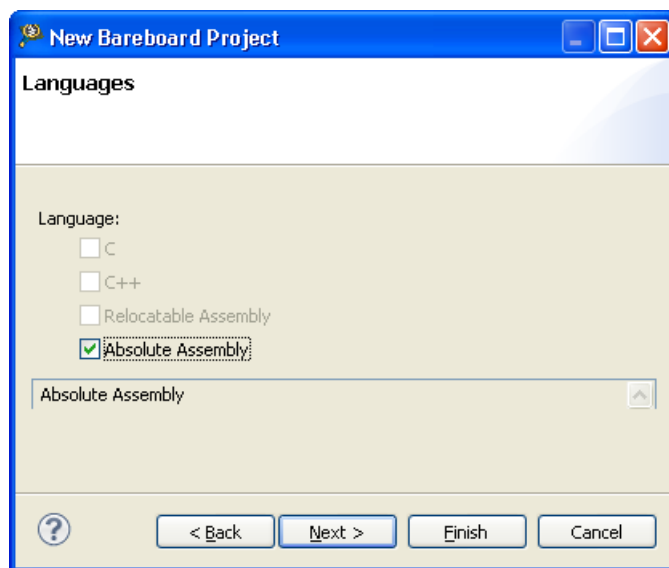


Figure 2-28. Language Page

- Click **Next**.

The **Rapid Application Development** page appears.

- Select the appropriate option.
- Click **Finish**.

The absolute assembly project is created and displayed in the **CodeWarrior Projects** view.

## 2.8.2 Adapting Absolute Assembly File Created by Wizard

Modify the absolute assembly file `main.asm` as the following listing displays.

The `ORG` directives must specify the absolute memory areas for ROM and RAM. The following listing shows an adaptation of the `main.asm` file produced previously by the Wizard. This file may be used by the Assembler build tool or IDE to directly generate an ABS file.

### Listing: Example Source File - main.asm

```

;*****
;* This stationery serves as the framework for a user *
;* application. For a more comprehensive program that *
;* demonstrates the more advanced functionality of this *
;* processor, please see the demonstration applications *
;* located in the examples subdirectory of CodeWarrior for *
;* Microcontrollers V10.x program directory. *
;*****
; application entry point
        ABSENTRY  _Startup
; export symbols
        XDEF  _Startup, main
        ; we use '_Startup' as an export symbol. This allows
        ; us to reference '_Startup' either in the linker
        ; *.prm file or from C/C++ later on.
        ; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
; variable/data section
        ORG  $0040
Counter:  DS.B  1
FiboRes:  DS.B  1
; initial value for SP
initStack: EQU  $023E
; code section
        ORG  $8000
main:
_Startup:
        LDHX  #initStack      ; initialize the stack pointer
        TXS
    
```

## Directly Generating ABS File

```

        CLI                ; enable interrupts

mainLoop:

        CLRA              ; A contains a counter.

cntLoop:   INCA

        CBEQA #14,mainLoop ; Larger values cause overflow.

        feed_watchdog    ;

        STA   Counter    ; update global

        BSR   CalcFibo

        STA   FiboRes    ; store result

        LDA   Counter

        BRA   cntLoop    ; next round

CalcFibo: ; Function to compute Fibonacci numbers. Argument is in A.

        DBNZA fiboDo     ; fiboDo

        INCA

        RTS

fiboDo:

        PSHA            ; the counter

        CLRX            ; second last = 0

        LDA   #$01      ; last = 1

FiboLoop: PSHA            ; push last

        TXA

        ADD   1,SP

        PULX

        DBNZ  1,SP,FiboLoop

FiboDone: PULH            ; release counter

        RTS            ; Result in A

;*****

;* spurious - Spurious Interrupt Service Routine.          *

;*           (unwanted interrupt)                          *

;*****

spurious:                ; Put here so the security

        NOP            ; value does not change

        RTI            ; all the time.

;*****

;*           Interrupt Vectors                            *

```



```

;*****
    ORG    $FFFA
    DC.W  spurious      ;
    DC.W  spurious      ; SWI
    DC.W  _Startup      ; Reset
    
```

The following listing is a similar example for RS08.

### Listing: Example Source File `abstest_rs08.asm`

```

ABSENTRY entry; Specifies the application Entry point
XDEF entry ; Make the symbol entry visible (needed for debugging)

    ORG $20 ; Define an absolute constant section
var1: DC.B 5 ; Assign 5 to the symbol var1

    ORG $40 ; Define an absolute data section
data: DS.B 1 ; Define one byte variable in RAM at $80

    ORG $3C00 ; Define an absolute code section
entry:
    LDA var1
main:
    INCA
    STA data
    BRA main
    
```

When writing your assembly source file for direct absolute file generation, pay special attention to the following points:

- The Reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application's entry point address is created at the reset vector address. To set the entry point of the application at address \$FFFA on the `_Startup` label the following code is needed.

#### Listing: Setting the Reset vector address

```

    ORG    $FFFA
    DC.W  spurious      ;
    DC.W  spurious      ; SWI
    DC.W  _Startup      ; Reset
    
```

The `ABSENTRY` directive is used to write the address of the application entry point in the generated absolute file. To set the entry point of the application on the `_Startup` label in the absolute file, the following code is needed.

#### Listing: Using `ABSENTRY` to enter the entry-point address

ABSENTRY \_Startup

### CAUTION

We strongly recommend that you use separate sections for code, (variable) data, and constants. All sections used in the assembler application must be absolute and defined using the `ORG` directive. The addresses for constant or code sections have to be located in the ROM memory area, while the data sections have to be located in a RAM area (according to the hardware that you intend to use). The programmer is responsible for making sure that no section overlaps occur.

## 2.8.3 Generating Absolute Assembly Using CodeWarrior IDE

To produce the executable `*.abs` file using the CodeWarrior IDE:

1. Select the absolute assembly project, `AbsoluteAssembly`, in the **CodeWarrior Projects** view.
2. Select **Project > Build Project**.

The CodeWarrior IDE produces the same `*.abs` output files that the Assembler and Linker generated for relocatable assembly.

The `*.sx` file generated in the `HCS08` folder of the project directory is a standard S-Record File. You can burn this file directly into a ROM memory.

## 2.8.4 Generating Absolute Assembly Using Assembler Build Tool

Use the same project, `Model T`, that was used for the relocatable assembly project. Modify the `main.asm` in the `Model T\Sources` folder as per the [Listing: Example Source File - main.asm](#).

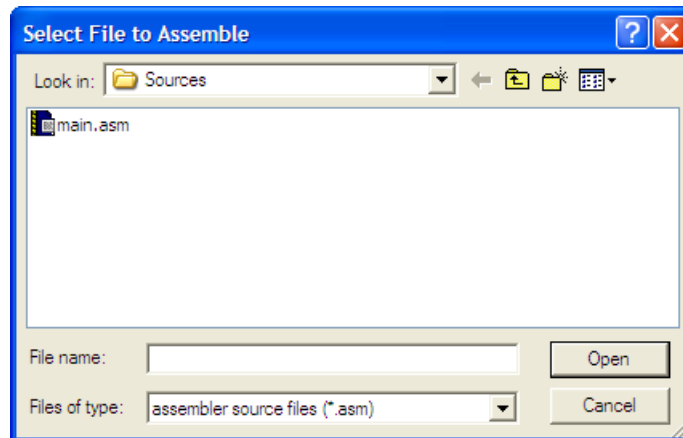
1. Start the Assembler by opening the `ahc08.exe` file in the `prog` folder in the `<CWInstallDir>\MCU` folder.

The *Assembler* window opens. Close the **Tip of the Day** dialog box.

2. Select **File > Load Configuration**. Browse for the project directory and set it as the current directory for the Assembler .

3. Select **Assembler > Options**. The **Option Settings** dialog box appears.
4. In the **Output** dialog box, check the **Object File Format** checkbox. A list box is displayed in the dialog box below the list of checkboxes.
5. Select the **ELF/DWARF 2.0 Absolute File** option from the list box. Click **OK**.
6. Select **File > Assemble**.

The **Select File to Assemble** dialog box appears, as the following image displays.



**Figure 2-29. Select File to Assemble Dialog Box**

7. Browse and select the absolute-assembly source-code file `main.asm`.
8. Click **Open**.

The Assembler now assembles the source code.

Make sure that the `GENPATH` configurations are set for the two include files needed for the `main.asm` file in this project in case they have not yet been previously set. Messages about the assembly process appears in the assembler main window.

The messages indicate that:

- An assembly source code (`main.asm`) file, plus `derivative.inc` and `mc9s08ac128.inc` files are read as input.
- A debugging (`main.dbg`) file is generated in the project directory.
- An S-Record File is created, `main.sx`. This file can be used to program ROM memory.
- An absolute executable file is generated, `main.abs`.
- The Code Size is 51 bytes.
- An assembly outlet listing file (`main.lst`) was written to the project directory.

## 2.9 Assembler Build Properties Panels

The following sections describe how to configure the [HCS08 Assembler Build Properties Panels](#) and [RS08 Assembler Build Properties Panels](#). These panels are part of the project's build properties settings, which are managed in the **Properties** dialog box. To access these panels, proceed as follows:

1. Select the project for which you want to set the build properties, in the **CodeWarrior Projects** view.
2. Select **Project > Properties**.

The **Properties for <project>** dialog box appears.

3. Expand the **C/C++ Build** tree node and select **Settings**.

The various settings for the build tools appears in the right panel.

If not, click the **Tool Settings** tab. If you have selected an **HCS08** project, the HCS08 tool settings page appears.

If you have selected an **RS08** project, the RS08 tool settings page appears.

The options are grouped by tool, such as **General** options, **Linker** options, **Assembler** options, and **Assembler** options. Depending on the build properties you wish to configure, select the appropriate option in the **Tool Settings** tab page.

## 2.9.1 HCS08 Assembler Build Properties Panels

The following listed are the build properties panels for the HC(S)08 Assembler.

### NOTE

For information about other build properties panels, refer to the *Microcontrollers V10.x Targeting Manual*.

**Table 2-1. Build Properties Panel for HC(S)08 Assembler**

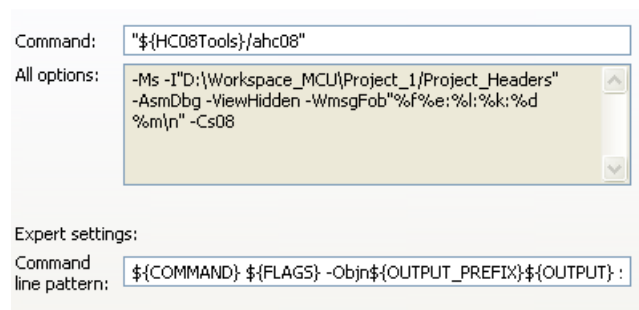
Build Tool	Build Properties Panels
HCS08 Assembler	<a href="#">HCS08 Assembler &gt; Output</a> <a href="#">HCS08 Assembler &gt; Output &gt; Configure listing file</a> <a href="#">HCS08 Assembler &gt; Input</a> <a href="#">HCS08 Assembler &gt; Language</a> <a href="#">HCS08 Assembler &gt; Language &gt; Compatibility modes</a> <a href="#">HCS08 Assembler &gt; Host</a> <a href="#">HCS08 Assembler &gt; Code Generation</a> <a href="#">HCS08 Assembler &gt; Messages</a> <a href="#">HCS08 Assembler &gt; Messages &gt; Disable user messages</a>

**Table 2-1. Build Properties Panel for HC(S)08 Assembler**

Build Tool	Build Properties Panels
	HCS08 Assembler > General

### 2.9.1.1 HCS08 Assembler

Use this panel to specify the command, options, and expert settings for the build tool assembler. The following image shows the **Assembler** settings.


**Figure 2-30. Tool Settings - Assembler**

The following table lists and describes the assembler options for HCS08.

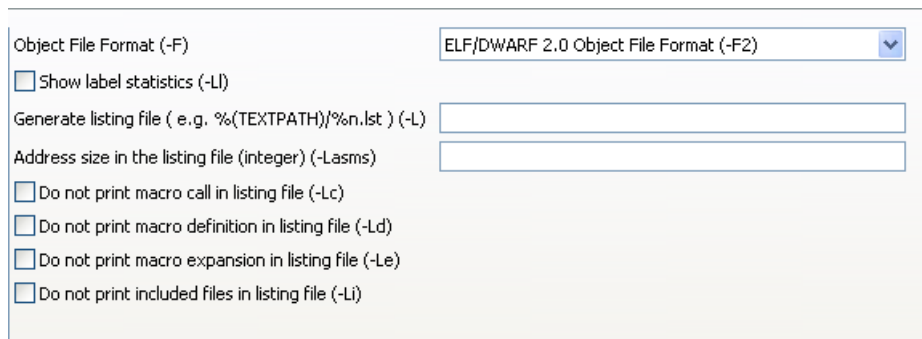
**Table 2-2. Tool Settings - Assembler Options**

Option	Description
Command	Shows the location of the assembler executable file. You can specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the assembler will be called with.
Expert Settings Command line pattern	Shows the expert settings command line parameters; default is <code>&amp;{COMMAND} &amp;{FLAGS} - Objn&amp;{OUTPUT_PREFIX}&amp;{OUTPUT} &amp;{INPUTS}</code> .

### 2.9.1.2 HCS08 Assembler > Output

Use this panel to control how the assembler generates the output file, as well as error and warning messages. You can specify whether to allocate constant objects in ROM, generate debugging information, and strip file path information.

The following image shows the **Output** panel.



**Figure 2-31. Tool Settings - HCS08 Assembler > Output**

The following table lists and describes the output options for HCS08 Assembler.

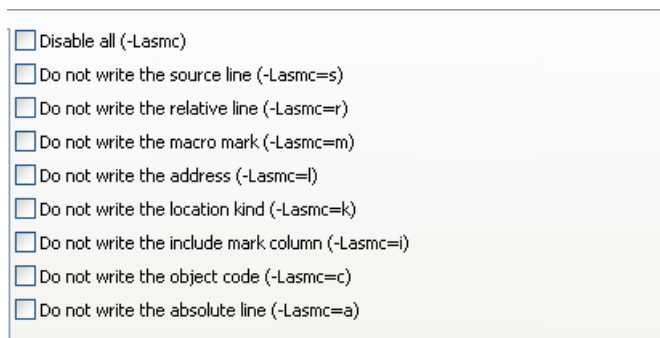
**Table 2-3. Tool Settings - HCS08 Assembler > Output Options**

Option	Description
Object File Format (-F)	Defines the format for the output file generated by the Assembler.
Show label statistics (-Ll)	Enables the assembler to append statistical information about the compilation session to the specified file. The information includes assembler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) for each procedure of the compiled file. The assembler appends the information to the specified filename (or the file make.txt, if no argument given). Set the TEXTPATH: Text File Path environment variable to store the file into the path specified by the environment variable. Otherwise the assembler stores the file in the current directory.
Generate listing file (for example, %(TEXTPATH)/%n.lst) (-L)	Specifies the name, %n, of the assembly listing file. The file is placed in the directory specified by %TEXTPATH. If this option is left blank, no listing file is output.
Address size in the listing file (integer) (-Lasm)	Specifies the size of the addresses displayed in the listing. Options are: <ul style="list-style-type: none"> <li>• 1 to display addresses as xx</li> <li>• 2 to display addresses as xxxx</li> <li>• 3 to display addresses as xxxxxx</li> <li>• 4 to display addresses asf xxxxxxxx</li> </ul>
Do not print macro call in listing file (-Lc)	Specifies whether macro calls encountered in the source code are expanded and appear in the listing file.
Do not print macro definition in listing file (-Ld)	Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.
Do not print macro expansion in listing file (-Le)	Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.
Do not print included files in listing file (-Li)	Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

### 2.9.1.3 HCS08 Assembler > Output > Configure listing file

Use this panel to specify the general assembler behavior.

The following image shows the **Configure listing file** panel.



**Figure 2-32. Tool Settings - HCS08 Assembler > Output > Configure listing file**

The following table lists and describes the configure listing file options for HCS08.

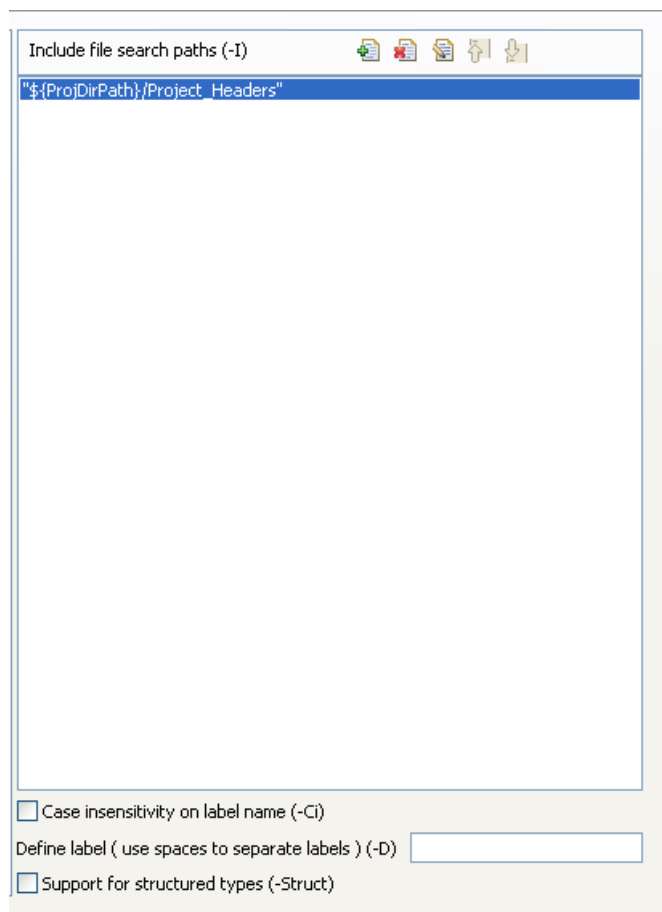
**Table 2-4. Tool Settings - Assembler > Output > Configure listing file Options**

Option	Description
Disable all (-Lasmc)	Print all the columns in the listing file.
Do not write the source line (-Lasmc=s)	Do not print source column in the listing file.
Do not write the relative line (-Lasmc=r)	Do not print relative column (Rel.) in the listing file.
Do not write the macro line (-Lasmc=m)	Do not print macro mark column in the listing file.
Do not write the address (-Lasmc=l)	Do not print address column (Loc) in the listing file.
Do not write the location kind (-Lasmc=k)	Do not print the location type column in the listing file.
Do not write the include mark column (-Lasmc=i)	Do not print the include mark column in the listing file.
Do not write the object code (-Lasmc=c)	Do not print the object code in the listing file.
Do not write the absolute line (-Lasmc=a)	Do not print the absolute column (Abs.) in the listing file.

### 2.9.1.4 HCS08 Assembler > Input

Use this panel to specify file search paths and any additional include files the **HCS08 Assembler** should use. You can specify multiple search paths and the order in which you want to perform the search.

The following image shows the **Input** panel.



**Figure 2-33. Tool Settings - HCS08 Assembler > Input**

The following table lists and describes the input options for HCS08 Assembler.

**Table 2-5. Tool Settings - HCS08 Assembler > Input Options**

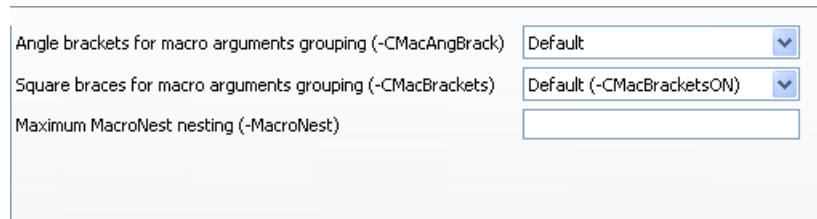
Option	Description
Include file search paths (-I)	Lists the included file search paths.
Case insensitivity on label name (-Ci)	Check to make the label names case insensitive.
Define label (use spaces to separate labels) (-D)	Define labels that have to be included in the RS08 assembler input.
Support for structured types (-Struct)	Check to include the support for structured types.

### 2.9.1.5 HCS08 Assembler > Language

Use this panel to specify code- and symbol-generation options for the HCS08 assembler.

The following image shows the **Language** panel.





**Figure 2-34. Tool Settings - HCS08 Assembler > Language**

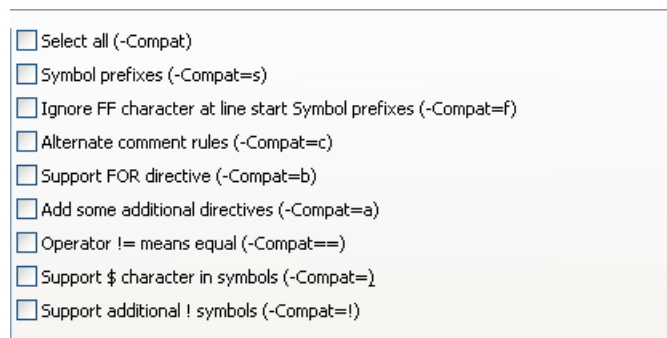
The following table lists and describes the language options for HCS08 Assembler.

**Table 2-6. Tool Settings - HCS08 Assembler > Language Options**

Option	Description
Angle brackets for macro arguments grouping (-CMacAngBrack)	Controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the [? ?] syntax instead. Options are: <ul style="list-style-type: none"> <li>• Allow</li> <li>• Disallow</li> </ul>
Square braces for macro arguments grouping (-CMacBrackets)	Controls the availability of the [? ?] syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for [? ?] in the macro invocation context. Options are: <ul style="list-style-type: none"> <li>• -CMacBracketsON</li> <li>• -CMacBracketsOFF</li> </ul>
Maximum MacroNest nesting (-MacroNest)	Controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

### 2.9.1.6 HCS08 Assembler > Language > Compatibility modes

The following image shows the **Compatibility modes** panel.



**Figure 2-35. Tool Settings - HCS08 Assembler > Language > Compatibility modes**

The following table lists and describes the compatibility mode options for HCS08 Assembler.

**Table 2-7. Tool Settings - HCS08 Assembler > Language > Compatibility mode Options**

Option	Description
Select all ( -Compat)	Check to enable all compatibility mode options.
Symbol prefixes ( -Compat=s)	With this suboption, the Assembler accepts "pgz:" and "byte:" prefixed for symbols in XDEFs and XREFs. They correspond to XREF.B or XDEF.B with the same symbols without the prefix.
Ignore FF character at line start Symbol prefixes ( -Compat=f)	With this suboption, an otherwise improper character recognized from feed character is ignored.
Alternate comment rules ( -Compat=c)	With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler does issue a warning if such a comment does not start with a "*" or a ";".
Support FOR directive ( -Compat=b)	With this suboption, the Assembler supports a FOR - Repeat assembly block assembly directive to generate repeated patterns more easily without having to use recursive macros.
Add some additional directives ( -Compat=a)	With this suboption, some additional directives are added for enhanced compatibility. The Assembler actually supports a SECT directive as an alias of the usual SECTION - Declare Relocatable Section assembly directive. The SECT directive takes the section name as its first argument.
Operator != means equal ( -Compat==)	The Assembler takes the default value of the != operator as not equal, as it is in the C language. For compatibility, this behavior can be changed to equal with this option. Because of the risks involved with this option for existing code, a message is issued for every != which is treated as equal.
Support \$ character in symbols ( -Compat=)	With this suboption, the Assembler supports to start identifiers with a \$ sign.
Support additional ! symbols ( -Compat=!)	<p>The following additional operators are defined when this option is used:</p> <ul style="list-style-type: none"> <li>• !^: exponentiation</li> <li>• !m: modulo</li> <li>• !@: signed greater or equal</li> <li>• !g: signed greater</li> <li>• !%: signed less or equal</li> <li>• !t: signed less than</li> <li>• !\$: unsigned greater or equal</li> <li>• !S: unsigned greater</li> <li>• !&amp;: unsigned less or equal</li> <li>• !l: unsigned less</li> <li>• !n: one complement</li> <li>• !w: low operator</li> <li>• !h: high operator</li> </ul> <p><b>Note:</b> The default values for the following ! operators are defined:</p> <ul style="list-style-type: none"> <li>• !.: binary AND</li> </ul>

**Table 2-7. Tool Settings - HCS08 Assembler > Language > Compatibility mode Options**

Option	Description
	<ul style="list-style-type: none"> <li>!x: exclusive OR</li> <li>!+: binary OR</li> </ul>

### 2.9.1.7 HCS08 Assembler > Host

Use this panel to specify the host settings of the HCS08.

The following image shows the Host settings.

**Figure 2-36. Tool Settings - HCS08 Assembler > Host**

The following table lists and describes the memory model options for HCS08.

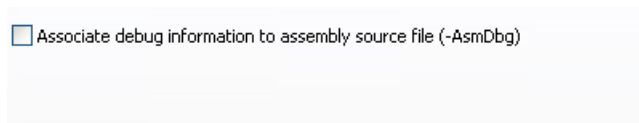
**Table 2-8. Tool Settings - HCS08 Assembler > Host Options**

Option	Description
Set environment variable (-Env)	This option sets an environment variable. Use this environment variable in the maker, or use to overwrite system environment variables.
Borrow license feature (-LicBorrow)	This option allows you 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.
Wait until a license is available from floating license server (-LicWait)	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.
Application Standard Occurrence	This option allows you to select the standard appearance for the application window. By default the option <b>-ViewHidden</b> is selected. For more Information, refer to the section <a href="#">-View: Application standard occurrence</a> .

### 2.9.1.8 HCS08 Assembler > Code Generation

Use this panel to specify the code generation assembler behavior.

The following image shows the **Code Generation** panel.



**Figure 2-37. Tool Settings - HCS08 Assembler > Code Generation**

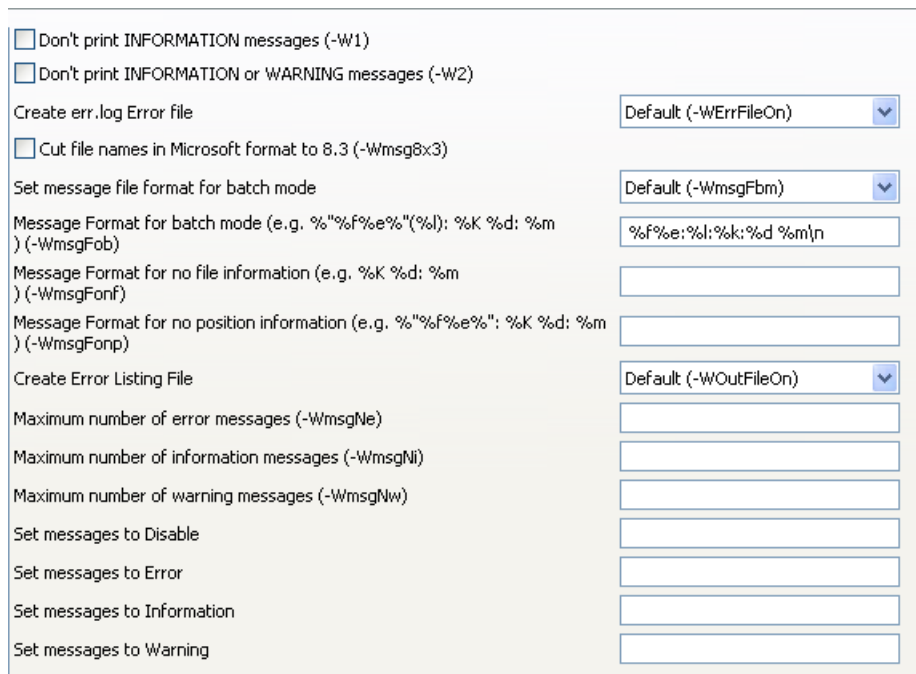
The following table lists and describes the code generation assembler options for HCS08.

**Table 2-9. Tool Settings - Assembler > Code Generation Options**

Option	Description
Associate debug information to assembly source file ( -AsmDbg)	Passes the assembly source file name information to DWARF sections. When the output .abs file is debugged, the actual assembly source file is displayed instead of intermediary <filename>.dbg file.

### 2.9.1.9 HCS08 Assembler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target. The following image shows the **Messages** panel.



**Figure 2-38. Tool Settings - HCS08 Assembler > Messages**

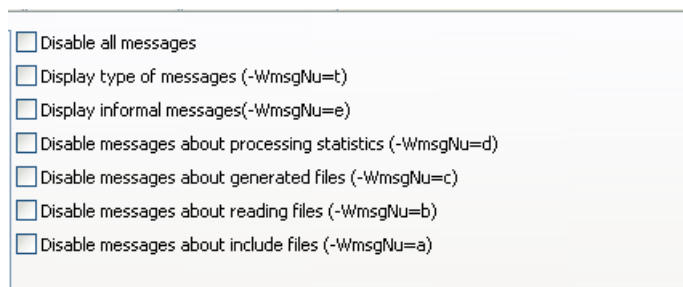
The following table lists and describes the message options.

**Table 2-10. Tool Settings - HCS08 Assembler > Messages Options**

Option	Description
Don't print INFORMATION messages (-W1)	Inhibits information message reporting. Only warning and error messages are generated.
Don't print INFORMATION or WARNING messages (-W2)	Suppresses all messages of type INFORMATION and WARNING. Only ERROR messages are generated.
Create err.log Error file	Using this option, the assembler uses a return code to report errors back to the tools. When errors occur, 16-bit window environments use err.log files, containing a list of error numbers, to report the errors. If no errors occur, the 16-bit window environments delete the err.log file.
Cut file names in Microsoft format to 8.3 (-Wmsg8x3)	Some editors (early versions of WinEdit) expect the filename in Microsoft message format (8.3 format). That means the filename can have up to eight characters and no more than a three-character extension. Longer filenames are possible when you use Win95 or WinNT. This option truncates the filename to the 8.3 format.
Set message file format for batch mode	Use this option to start the assembler with additional arguments (for example, files and assembler options). If you start the assembler with arguments (for example, from the Make Tool or with the '%f' argument from the CodeWright IDE), the assembler compiles the files in a batch mode. No assembler window is visible and the assembler terminates after job completion.
Message Format for batch mode (e.g. %"%f%e%"(%l): %K %d: %m) (-WmsgFob)	Specify additional command line options; type in custom flags that are not otherwise available in the UI. Default value is %f %e: %l: %k: %d %m\n
Message Format for no file information (e.g. %K %d: %m) (-WmsgFonf)	If there is no file information available for a message, then <string> defines the message format string to use.
Message Format for no position information (e.g. %"%f%e%": %K %d: %m) (-WmsgFonp)	If there is no position information available for a message, then <string> defines the message format string to use.
Create Error Listing File	This option controls whether the assembler creates an error listing file. The error listing file contains a list of all messages and errors that occur during processing.
Maximum number of error messages (-WmsgNe)	Specify the number of errors allowed until the application stops processing.
Maximum number of information messages (-WmsgNi)	Specify the maximum number of information messages allowed.
Maximum number of warning messages (-WmsgNw)	Specify the maximum number of warnings allowed.
Set messages to Disable	Enter the messages that you want to disable.
Set messages to Error	Enter the messages that you want to set as error.
Set messages to Information	Enter the messages that you want to set as information.
Set messages to Warning	Enter the messages that you want to set as warning.

### 2.9.1.10 HCS08 Assembler > Messages > Disable user messages

Use this panel to specify the options for disabling the user messages for the HC(S)08 assembler. The following image shows the **Disable user messages** panel.



**Figure 2-39. Tool Settings - HCS08 Assembler > Messages > Disable user messages**

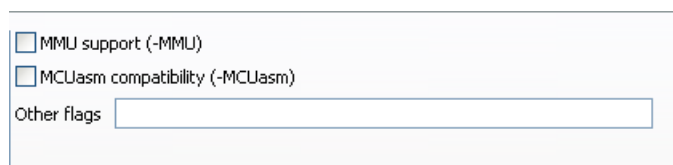
#### NOTE

For information about the options available in the **Disable user messages** panel of HC(S)08 assembler, refer to the section - [WmsgNu: Disable user messages](#).

### 2.9.1.11 HCS08 Assembler > General

Use this panel to specify the general assembler behavior.

The following image shows the **General** panel.



**Figure 2-40. Tool Settings - HCS08 Assembler > General**

The following table lists and describes the general assembler options for HCS08.

**Table 2-11. Tool Settings - Assembler > General Options**

Option	Description
MMU Support (-MMU)	Check to inform the assembler that CALL and RTC instructions are available, enabling code banking, and that the current architecture has extended data access capabilities, enabling support for <code>__linear</code> data types. This option can be used only when <code>-Cs08</code> is enabled.

*Table continues on the next page...*

**Table 2-11. Tool Settings - Assembler > General Options (continued)**

Option	Description
MCUasm compatibility ( -MCUasm)	Check to activate the compatibility mode with the MCUasm Assembler.
Other flags	Specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.

## 2.9.2 RS08 Assembler Build Properties Panels

The following table lists the build properties panels for the RS08 Assembler.

### NOTE

For information about other build properties panels, refer to the *Microcontrollers V10.x Targeting Manual*.

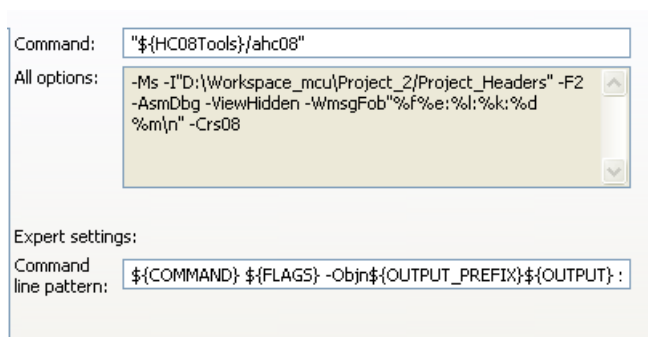
**Table 2-12. Build Properties for RS08 Assembler**

Build Tool	Build Properties Panels
RS08 Assembler	RS08 Assembler > Output
	RS08 Assembler > Output > Configure Listing File
	RS08 Assembler > Input
	RS08 Assembler > Language
	RS08 Assembler > Language > Compatibility modes
	RS08 Assembler > Host
	RS08 Assembler > Code Generation
	RS08 Assembler > Messages
	RS08 Assembler > Messages > Disable user messages
	RS08 Assembler > General

### 2.9.2.1 RS08 Assembler

Use this panel to specify the command, options, and expert settings for the build tool assembler.

The following image shows the **Assembler** settings.


**Figure 2-41. Tool Settings - RS08 Assembler**

The following table lists and describes the assembler options for RS08.

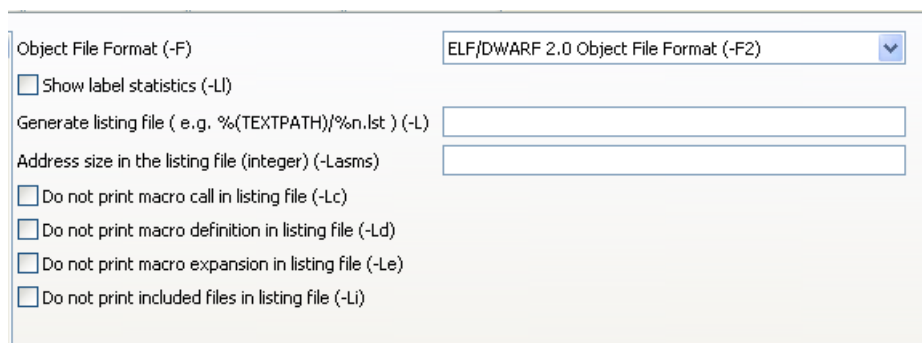
**Table 2-13. Tool Settings - RS08 Assembler Options**

Option	Description
Command	Shows the location of the assembler executable file. Default value is: " \${HC08Tools}/ahc08.exe". You can specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the assembler will be called with.
Expert Settings Command line pattern	Shows the expert settings command line parameters; default is \${COMMAND} \${COMMAND} \${FLAGS} -Objn\${OUTPUT_PREFIX}\${OUTPUT} \${INPUTS}.

### 2.9.2.2 RS08 Assembler > Output

Use this panel to control how the assembler generates the output file, as well as error and warning messages. You can specify whether to allocate constant objects in ROM, generate debugging information, and strip file path information.

The following image shows the **Output** panel.


**Figure 2-42. Tool Settings - RS08 Assembler > Output**



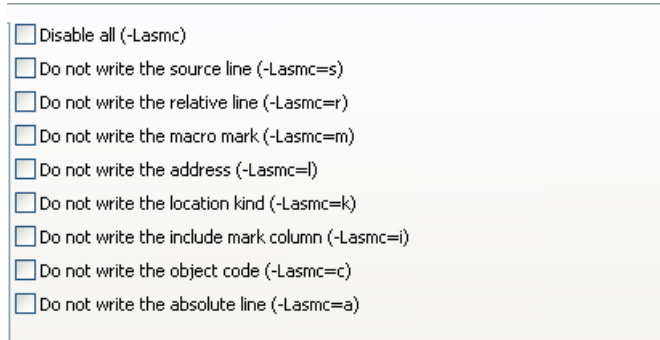
The following table lists and describes the output options for RS08 Assembler.

**Table 2-14. Tool Settings - RS08 Assembler > Output Options**

Option	Description
Object File Format (-F)	Defines the format for the output file generated by the Assembler.
Show label statistics (-Li)	Using the -Ll option, the assembler appends statistical information about the compilation session to the specified file. The information includes assembler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) for each procedure of the compiled file. The assembler appends the information to the specified filename (or the file make.txt, if no argument given). Set the TEXTPATH: Text File Path environment variable to store the file into the path specified by the environment variable. Otherwise the assembler stores the file in the current directory.
Generate listing file ( e.g. %(TEXTPATH)/%n.lst ) (-L)	The -Lasm option causes the assembler to generate an assembler listing file directly. The assembler also prints all assembler-generated instructions to this file. The option specifies the name of the file. If no name is specified, the assembler takes a default of %n.lst. If the resulting filename contains no path information the assembler uses the TEXTPATH: Text File Path environment variable. The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, use this option only to review the generated code. It cannot currently be used to generate a file for assembly.
Address size in the listing file (-Lasms)	Specifies the size of the addresses displayed in the listing. Options are: <ul style="list-style-type: none"> <li>• 1 to display addresses as xx</li> <li>• 2 to display addresses as xxxx</li> <li>• 3 to display addresses as xxxxxx</li> <li>• 4 to display addresses asf xxxxxxxx</li> </ul>
Do not print macro call in listing file (-Lc)	Specifies whether macro calls encountered in the source code are expanded and appear in the listing file.
Do not print macro definition in listing file (-Ld)	Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.
Do not print macro expansion in listing file (-Le)	Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.
Do not print included files in listing file (-Li)	Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

### 2.9.2.3 RS08 Assembler > Output > Configure Listing File

Use this panel to configure the listing file options of RS08 assembler. The following image shows the **Configure Listing File** panel.



**Figure 2-43. Tool Settings - RS08 Assembler > Output > Configure listing file**

The following table lists and describes the Configure Listing File options for RS08 Assembler.

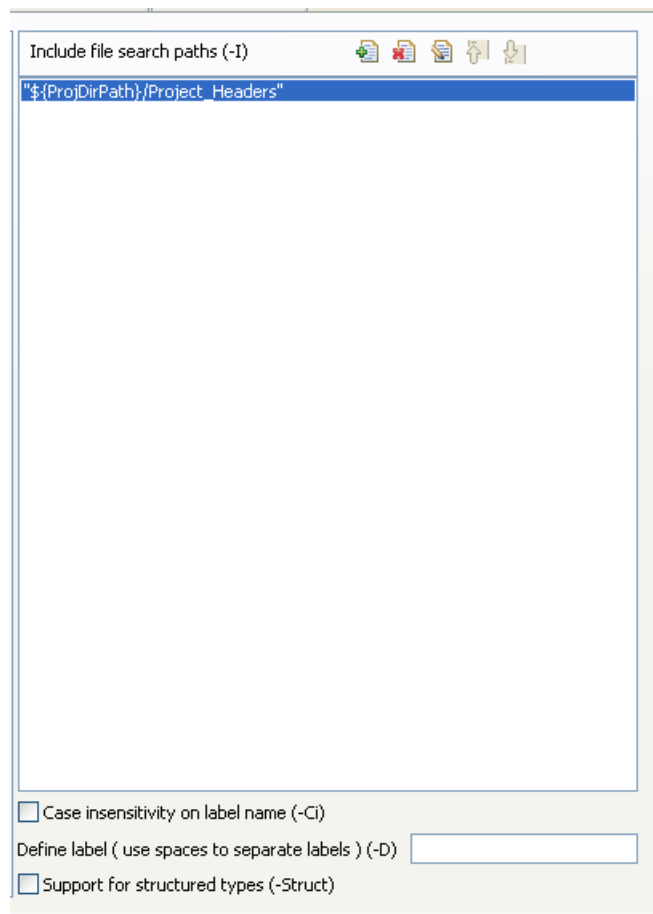
**Table 2-15. Tool Settings - RS08 Assembler > Configure listing file Options**

Option	Description
Disable all (-Lasmc)	Print all the columns in the listing file.
Do not write the source line (-Lasmc=s)	Do not print source column in the listing file.
Do not write the relative line (-Lasmc=r)	Do not print relative column (Rel.) in the listing file.
Do not write the macro mark (-Lasmc=m)	Do not print macro mark column in the listing file.
Do not write the address (-Lasmc=l)	Do not print address column (Loc) in the listing file.
Do not write the location kind (-Lasmc=k)	Do not print the location type column in the listing file.
Do not write the include mark column (-Lasmc=i)	Do not print the include mark column in the listing file.
Do not write the object code (-Lasmc=c)	Do not print the object code in the listing file.
Do not write the absolute line (-Lasmc=a)	Do not print the absolute column (Abs.) in the listing file.

### 2.9.2.4 RS08 Assembler > Input

Use this panel to specify file search paths and any additional include files the **RS08 Assembler** should use. You can specify multiple search paths and the order in which you want to perform the search.

The following image shows the **Input** panel.



**Figure 2-44. Tool Settings - RS08 Assembler > Input**

The following table lists and describes the input options of RS08 assembler.

**Table 2-16. Tool Settings - Assembler > Input options**

Button	Description
Include file search paths (-I)	Lists the included file search paths.
Case insensitivity on label name (-Ci)	Check to make the label names case insensitive.
Define label (use spaces to separate labels) (-D)	Define labels that have to be included in the RS08 assembler input.
Support for structured types (-Struct)	Check to include the support for structured types.

The following table lists and describes the toolbar buttons that help work with the file search paths.

**Table 2-17. Search Paths Toolbar Buttons**

Button	Description
	<b>Add</b> - Click to open the <b>Add directory path</b> dialog box and specify the file search path.

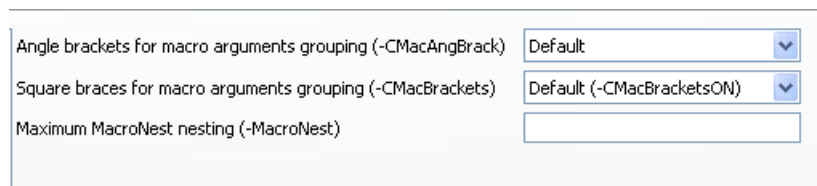
*Table continues on the next page...*

**Table 2-17. Search Paths Toolbar Buttons (continued)**

Button	Description
	<b>Delete</b> - Click to delete the selected file search path.
	<b>Edit</b> - Click to open the <b>Edit directory path</b> dialog box and update the selected object file search path.
	<b>Move up</b> - Click to move the selected file search path one position higher in the list.
	<b>Move down</b> - Click to move the selected file search path one position lower in the list.

### 2.9.2.5 RS08 Assembler > Language

Use this panel to specify code- and symbol-generation options for the RS08 assembler. The following image shows the **Language** panel.



**Figure 2-45. Tool Settings - RS08 Assembler > Language**

The following table lists and describes the language options for RS08 Assembler.

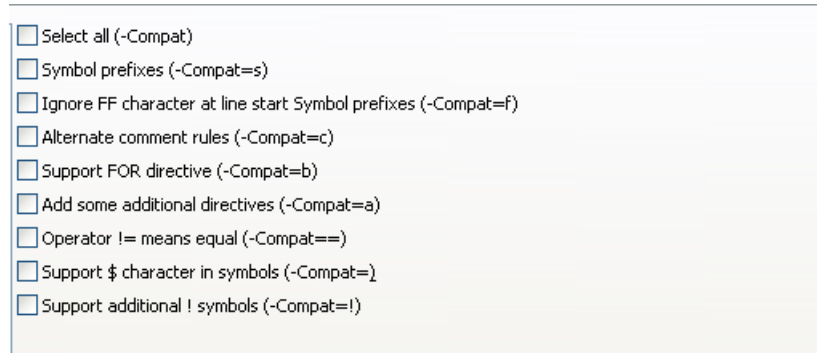
**Table 2-18. Tool Settings - RS08 Assembler > Language Options**

Option	Description
Angle brackets for macro arguments grouping (-CMacAngBrack)	Controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the [? ?] syntax instead. Options are: <ul style="list-style-type: none"> <li>• Allow</li> <li>• Disallow</li> </ul>
Square braces for macro arguments grouping (-CMacBrackets)	Controls the availability of the [? ?] syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for [?] in the macro invocation context. Options are: <ul style="list-style-type: none"> <li>• -CMacBracketsON</li> <li>• -CMacBracketsOFF</li> </ul>
Maximum MacroNest nesting (-MacroNest)	Controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

### 2.9.2.6 RS08 Assembler > Language > Compatibility modes

Use this panel to specify the compatibility modes options of the RS08 assembler.

The following image shows the **Compatibility modes** panel.



**Figure 2-46. Tool Settings - RS08 Assembler > Compatibility modes**

The following table lists and describes the compatibility mode options for RS08 Assembler.

**Table 2-19. Tool Settings - RS08 Assembler > Compatibility modes Options**

Option	Description
Select all (-Compat)	Check to enable all compatibility mode options.
Symbol prefixes (-Compat=s)	With this suboption, the Assembler accepts "pgz:" and "byte:" prefixed for symbols in XDEFs and XREFs. They correspond to XREF.B or XDEF.B with the same symbols without the prefix.
Ignore FF character at line start Symbol prefixes (-Compat=f)	With this suboption, an otherwise improper character recognized from feed character is ignored.
Alternate comment rules (-Compat=c)	With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler does issue a warning if such a comment does not start with a "*" or a ";".
Support FOR directive (-Compat=b)	With this suboption, the Assembler supports a FOR - Repeat assembly block assembly directive to generate repeated patterns more easily without having to use recursive macros.
Add some additional directives (-Compat=a)	With this suboption, some additional directives are added for enhanced compatibility. The Assembler actually supports a SECT directive as an alias of the usual SECTION - Declare Relocatable Section assembly directive. The SECT directive takes the section name as its first argument.

*Table continues on the next page...*

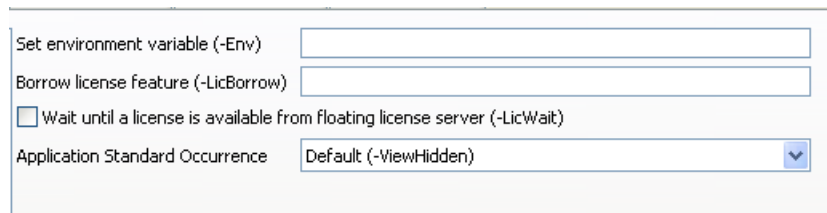
**Table 2-19. Tool Settings - RS08 Assembler > Compatibility modes Options (continued)**

Option	Description
Operator != means equal (-Compat==)	The Assembler takes the default value of the != operator as not equal, as it is in the C language. For compatibility, this behavior can be changed to equal with this option. Because of the risks involved with this option for existing code, a message is issued for every != which is treated as equal.
Support \$ character in symbols (-Compat=)	With this suboption, the Assembler supports to start identifiers with a \$ sign.
Support additional ! symbols (-Compat=!)	<p>The following additional operators are defined when this option is used:</p> <ul style="list-style-type: none"> <li>• !^: exponentiation</li> <li>• !m: modulo</li> <li>• !@: signed greater or equal</li> <li>• !g: signed greater</li> <li>• !%: signed less or equal</li> <li>• !t: signed less than</li> <li>• !\$: unsigned greater or equal</li> <li>• !S: unsigned greater</li> <li>• !&amp;: unsigned less or equal</li> <li>• !!: unsigned less</li> <li>• !n: one complement</li> <li>• !w: low operator</li> <li>• !h: high operator</li> </ul> <p><b>NOTE:</b> The default values for the following ! operators are defined:</p> <ul style="list-style-type: none"> <li>• !.: binary AND</li> <li>• !x: exclusive OR</li> <li>• !+: binary OR</li> </ul>

### 2.9.2.7 RS08 Assembler > Host

Use this panel to specify the host settings of the RS08 assembler.

The following image shows the **Host** settings.



**Figure 2-47. Tool Settings - Host**

The following table lists and describes the memory model options for RS08.

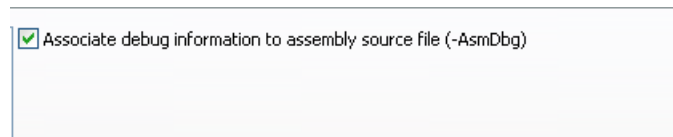
**Table 2-20. Tool Settings - RS08 Assembler > Host Options**

Option	Description
Set environment variable (-Env)	This option sets an environment variable. Use this environment variable in the maker, or use to overwrite system environment variables.
Borrow license feature (-LicBorrow)	This option allows you 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.
Wait until a license is available from floating license server (-LicWait)	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.
Application Standard Occurrence	This option allows you to select the standard appearance for the application window. By default the option <b>-ViewHidden</b> is selected. For more Information, refer to the section <a href="#">-View: Application standard occurrence</a> .

### 2.9.2.8 RS08 Assembler > Code Generation

Use this panel to specify the code generation options of the RS08 assembler.

The following image shows the **Code Generation** panel.



**Figure 2-48. Tool Settings - RS08 Assembler > Code Generation**

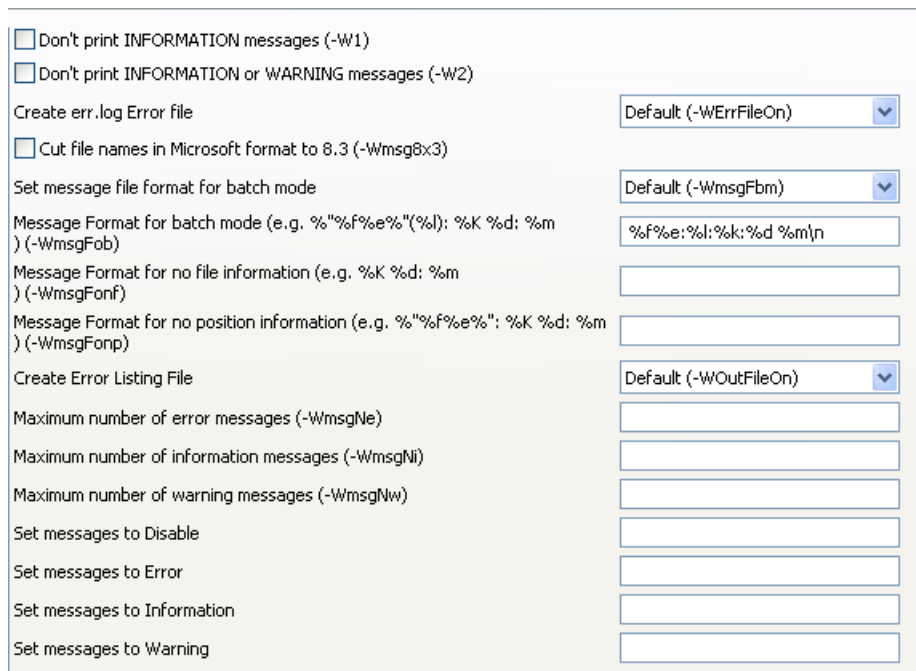
The following table lists and describes the Code Generation options for RS08 Assembler.

**Table 2-21. Tool Settings - RS08 Assembler > Code Generation Options**

Option	Description
Associate debug information to assembly source file (-Asmdbg)	Passes the assembly source file name information to DWARF sections. When the output .abs file is debugged, the actual assembly source file is displayed instead of intermediary <filename>.dbg file.

### 2.9.2.9 RS08 Assembler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target. The following image shows the **Messages** panel.



**Figure 2-49. Tool Settings - RS08 Assembler > Messages**

The following table lists and describes the message options.

**Table 2-22. Tool Settings - RS08 Assembler > Messages Options**

Option	Description
Don't print INFORMATION messages ( -W1)	Inhibits information message reporting. Only warning and error messages are generated.
Don't print INFORMATION or WARNING messages ( -W2)	Suppresses all messages of type INFORMATION and WARNING. Only ERROR messages are generated.
Create err.log Error file	Using this option, the assembler uses a return code to report errors back to the tools. When errors occur, 16-bit window environments use err.log files, containing a list of error numbers, to report the errors. If no errors occur, the 16-bit window environments delete the err.log file.
Cut file names in Microsoft format to 8.3 ( -Wmsg8x3)	Some editors (early versions of WinEdit) expect the filename in Microsoft message format (8.3 format). That means the filename can have up to eight characters and no more than a three-character extension. Longer filenames are possible when you use Win95 or WinNT. This option truncates the filename to the 8.3 format.
Set message file format for batch mode	Use this option to start the assembler with additional arguments (for example, files and assembler options). If you start the assembler with arguments (for example, from the Make Tool or with the '%f' argument from the CodeWright

*Table continues on the next page...*

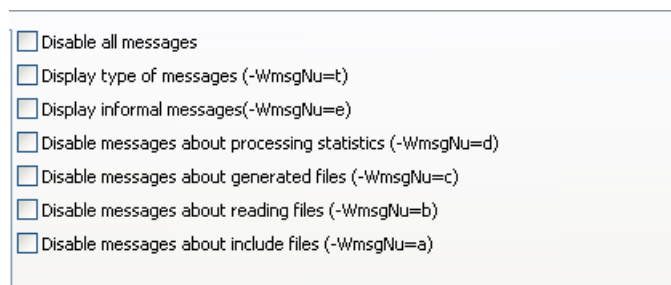


**Table 2-22. Tool Settings - RS08 Assembler > Messages Options (continued)**

Option	Description
	IDE), the assembler compiles the files in a batch mode. No assembler window is visible and the assembler terminates after job completion.
Message Format for batch mode (e.g. %"%f%e%"(%l): %K %d: %m) (-WmsgFob)	Specify additional command line options; type in custom flags that are not otherwise available in the UI. Default value is %f %e:%l:%k:%d %m\n
Message Format for no file information (e.g. %K %d: %m) (-WmsgFonf)	If there is no file information available for a message, then <string> defines the message format string to use.
Message Format for no position information (e.g. %"%f%e%": %K %d: %m) (-WmsgFonp)	If there is no position information available for a message, then <string> defines the message format string to use.
Create Error Listing File	This option controls whether the assembler creates an error listing file. The error listing file contains a list of all messages and errors that occur during processing.
Maximum number of error messages (-WmsgNe)	Specify the number of errors allowed until the application stops processing.
Maximum number of information messages (-WmsgNi)	Specify the maximum number of information messages allowed.
Maximum number of warning messages (-WmsgNw)	Specify the maximum number of warnings allowed.
Set messages to Disable	Enter the messages that you want to disable.
Set messages to Error	Enter the messages that you want to set as error.
Set messages to Information	Enter the messages that you want to set as information.
Set messages to Warning	Enter the messages that you want to set as warning.

### 2.9.2.10 RS08 Assembler > Messages > Disable user messages

Use this panel to specify the options for disabling the user messages for the RS08 assembler. The following image shows the **Disable user messages** panel.


**Figure 2-50. Tool Settings - RS08 Assembler > Messages > Disable user messages**

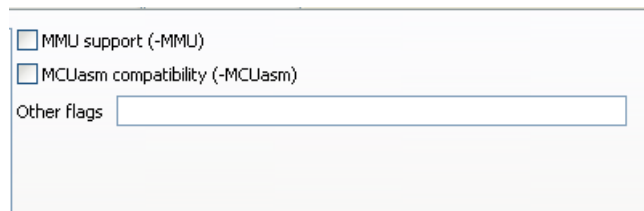
**NOTE**

For information about the options available in the **Disable user messages** panel of RS08 assembler, refer to the [-WmsgNu: Disable user messages](#).

**2.9.2.11 RS08 Assembler > General**

Use this panel to specify the general assembler behavior.

The following image shows the **General** panel.



**Figure 2-51. Tool Settings - RS08 Assembler > General**

The following table lists and describes the general assembler options for RS08.

**Table 2-23. Tool Settings - Assembler > General Options**

Option	Description
MMU Support (-MMU)	Check to inform the assembler that CALL and RTC instructions are available, enabling code banking, and that the current architecture has extended data access capabilities, enabling support for <code>__linear</code> data types. This option can be used only when <code>-Cs08</code> is enabled.
MCUasm compatibility (-MCUasm)	Check to activate the compatibility mode with the MCUasm Assembler.
Other Flags	Specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.

## Chapter 3

# Assembler Graphical User Interface

The Macro Assembler runs under *Windows® 2000, Windows XP, Windows Vista™, and compatible operating systems.*

This chapter covers the following topics:

- [Starting Assembler](#)
- [Assembler Main Window](#)
- [Editor Setting Dialog Box](#)
- [Save Configuration Dialog Box](#)
- [Option Settings Dialog Box](#)
- [Message Settings Dialog Box](#)
- [About Dialog Box](#)
- [Specifying Input File](#)
- [Message/Error Feedback](#)

### 3.1 Starting Assembler

When you start the Assembler, the Assembler displays a standard **Tip of the Day** dialog box containing news and tips about the Assembler.



**Figure 3-1. Tip of the Day dialog box**

To use the **Tip of the Day** dialog box:

- Click **Next Tip** to see the next piece of information about the Assembler.
- Click **Close** to close the **Tip of the Day** dialog box.
- If you do not want the Assembler to automatically open the standard **Tip of the Day** dialog box when the Assembler is started, clear the **Show Tips on StartUp** checkbox.
- If you want the Assembler to automatically open the standard **Tip of the Day** dialog box at Assembler start up, select **Help > Tip of the Day** . The Assembler displays the **Tip of the Day** dialog box. Check the **Show Tips on StartUp** checkbox.

## 3.2 Assembler Main Window

This window is only visible on the screen when you do not specify any filename when you start the Assembler.

The Assembler window consists of a window title, a menu bar, a toolbar, a content area, and a status bar.

### 3.2.1 Window Title

The window title displays the Assembler name and the project name. If a project is not loaded, the Assembler displays *Default Configuration* in the window title. An asterisk ( \*) after the configuration name indicates that some settings have changed. The Assembler adds an asterisk ( \*) whenever an option, the editor configuration, or the window appearance changes.

### 3.2.2 Content Area

The Assembler displays logging information about the assembly session in the content area. This logging information consists of:

- the name of the file being assembled,
- the whole name (including full path specifications) of the files processed (main assembly file and all included files),
- the list of any error, warning, and information messages generated, and
- the size of the code (in bytes) generated during the assembly session.

When a file is dropped into the assembly window content area, the Assembler either loads the corresponding file as a configuration file or the Assembler assembles the file. The Assembler loads the file as a configuration if the file has the \*.ini extension. If the file does not end with the \*.ini extension, the Assembler assembles the file using the current option settings.

All text in the assembler window content area can have context information consisting of two items:

- a filename including a position inside of a file and
- a message number.

File context information is available for all output lines where a filename is displayed. There are two ways to open the file specified in the file-context information in the editor specified in the editor configuration:

- If a file context is available for a line, double-click on a line containing file-context information.
- Click with the right mouse on the line and select **Open** . This entry is only available if a file context is available.

If the Assembler cannot open a file even though a context menu entry is present, then the editor configuration information is incorrect (refer to the [Editor Setting Dialog Box](#) section below).

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 the **F1** key. If the selected line does not have a message number, the main help is displayed.
- Press **Shift-F1** and then click on the message text. If the point clicked does not have a message number, the main help is displayed.
- Click the right mouse button on the message text and select **Help on** . This entry is only available if a message number is available.

### 3.2.3 Toolbar

The three buttons on the left hand side of the toolbar correspond to the menu items of the **File** menu. You can use the **New**, **Load**, and **Save** buttons to reset, load and save configuration files for the Macro Assembler.

The **Help** button and the **Context Help** button allow you to open the *Help* file or the *Context Help*.

When pressing the buttons above, the mouse cursor changes to a question mark beside an arrow. The Assembler opens Help for the next item on which you click. You can get specific Help on menus, toolbar buttons, or on the window area by using this *Context Help*.

The editable combo box contains a list of the last commands which were executed. After a command line has been selected or entered in this combo box, click the **Assemble** button to execute this command. The Stop button becomes enabled whenever some file is assembled. When the **Stop** button is pressed, the assembler stops the assembly process.

Pressing the **Options Dialog Box** button opens the **Option Settings** dialog box.

Pressing the **Message Dialog Box** button opens the **Message Settings** dialog box.

Pressing the **Clear** button clears the assembler window's content area.

### 3.2.4 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 to which you are pointing.

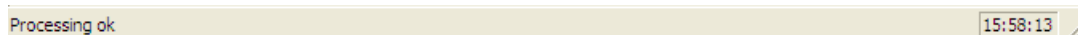


Figure 3-2. Status Bar

## 3.2.5 Assembler Menu Bar

The following table lists the menus available in the menu bar:

**Table 3-1. Menu bar options**

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

## 3.2.6 File Menu

With the **File** menu, Assembler configuration files can be saved or loaded. An Assembler configuration file contains the following information:

- the assembler option settings specified in the assembler dialog boxes,
- the list of the last command line which was executed and the current command line,
- the window position, size, and font,
- the editor currently associated with the Assembler. This editor may be specifically associated with the Assembler or globally defined for all *Tools* (see the [Editor Setting Dialog Box](#)),
- the *Tips of the Day* settings, including its startup configuration, and what is the current entry, and
- Configuration files are text files which have the standard \* .ini extension. You can define as many configuration files as required for the project and can switch among the different configuration files using the **File > Load Configuration**, **File > Save Configuration** menu entries, or the corresponding toolbar buttons.

**Table 3-2. File Menu Options**

Menu Entry	Description
Assemble	A standard <b>Open File</b> dialog box is opened, displaying the list of all the * .asm files in the project directory. The input file can be selected using the features from the standard <b>Open File</b> dialog box. The selected file is assembled when the <b>Open File</b> dialog box is closed by clicking <b>OK</b> .

*Table continues on the next page...*

**Table 3-2. File Menu Options (continued)**

Menu Entry	Description
New/Default Configuration	Resets the Assembler option settings to their default values. The default Assembler options which are activated are specified in the <a href="#">Assembler Options</a> chapter.
Load Configuration	A standard <b>Open File</b> dialog box is opened, displaying the list of all the *.ini files in the project directory. The configuration file can be selected using the features from the standard <b>Open File</b> dialog box. The configuration data stored in the selected file is loaded and used in further assembly sessions.
Save Configuration	Saves the current settings in the configuration file specified on the title bar.
Save Configuration As...	A standard <b>Save As</b> dialog box is opened, displaying the list of all the *.ini files in the project directory. The name or location of the configuration file can be specified using the features from the standard <b>Save As</b> dialog box. The current settings are saved in the specified configuration file when the <b>Save As</b> dialog box is closed by clicking <b>OK</b> .
Configuration...	Opens the <b>Configuration</b> dialog box to specify the editor used for error feedback and which parts to save with a configuration. See <a href="#">Editor Setting Dialog Box</a> and <a href="#">Save Configuration Dialog Box</a> .
1. .... project.ini 2. ....	Recent project list. This list can be used to reopen a recently opened project.
Exit	Closes the Assembler.

### 3.2.7 Assembler Menu

The **Assembler** menu allows you to customize the Assembler. You can graphically set or reset the Assembler options or to stop the assembling process. The following table lists the assembler menu options.

**Table 3-3. Assembler Menu Options**

Menu entry	Description
Options	Defines the options which must be activated when assembling an input file (see <a href="#">Option Settings Dialog Box</a> ).
Messages	Maps messages to a different message class (see <a href="#">Message Settings Dialog Box</a> ).
Stop assembling	Stops the assembling of the current source file.



## 3.2.8 View Menu

The **View** menu lets you customize the Assembler window. You can specify if the status bar or the toolbar must be displayed or be hidden. You can also define the font used in the window or clear the window. The following table lists the **View** menu options.

**Table 3-4. View Menu Options**

Menu Entry	Description
Toolbar	Switches display from the toolbar in the Assembler window.
Status Bar	Switches display from the status bar in the Assembler window.
Log...	Customizes the output in the Assembler window content area. The following two entries in this table are available when you select Log:
Change Font	Opens a standard font dialog box. The options selected in the font dialog box are applied to the Assembler window content area.
Clear Log	Clears the Assembler window content area.

## 3.3 Editor Setting Dialog Box

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

These are the main entries for the Editor configuration:

- [Global Editor \(shared by all tools and projects\)](#)
- [Local Editor \(shared by all tools\)](#)
- [Editor Started with Command Line](#)
- [Editor Started with DDE](#)
- [CodeWarrior with COM](#)

### 3.3.1 Global Editor (shared by all tools and projects)

This entry (refer to the image listed below) is shared by all tools for all projects. This setting is stored in the [Editor] section of the `mcutools.ini` global initialization file. Some **Modifiers** can be specified in the editor command line.

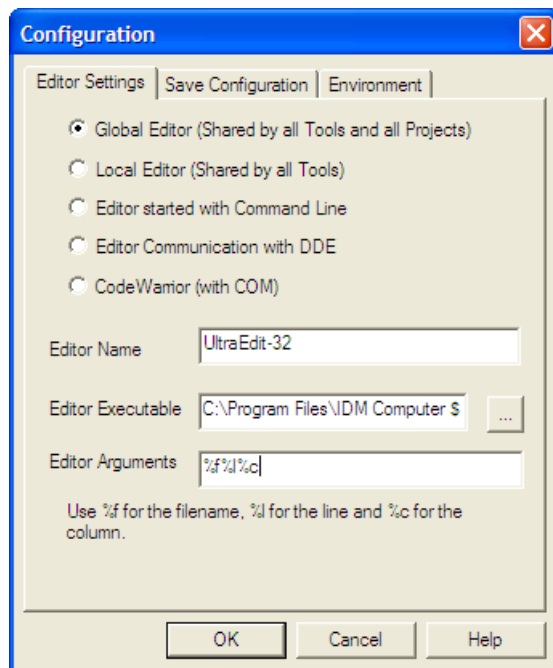


Figure 3-3. Global Editor Configuration Dialog Box

### 3.3.2 Local Editor (shared by all tools)

This entry is shared by all tools for the current project. This setting is stored in the [Editor] section of the local initialization file, usually `project.ini` in the current directory. Some **Modifiers** can be specified in the editor command line.

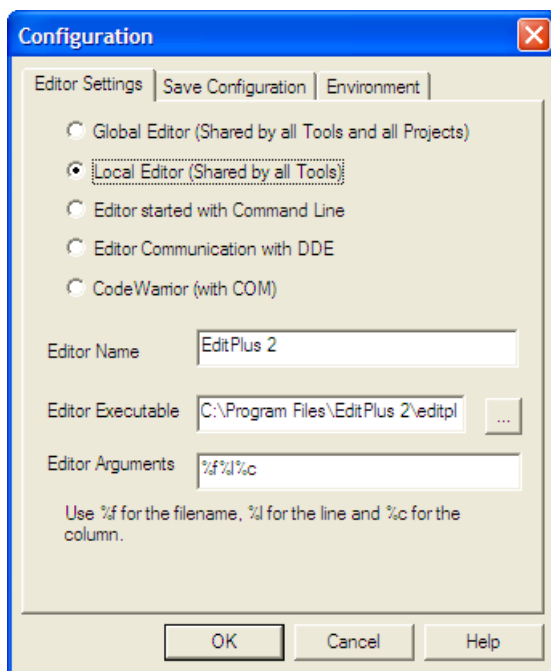


Figure 3-4. Local Editor Configuration Dialog Box

### 3.3.3 Editor Started with Command Line

When this editor type is selected, a separate editor is associated with the Assembler for error feedback. The editor configured in the shell is not used for error feedback.

Enter the command which should be used to start the editor (refer to the figure listed below).

The format from the editor command depends on the syntax which should be used to start the editor. Modifiers can be specified in the editor command line to refer to a filename and line and column position numbers. (for more information, refer to the [Modifiers](#) section)

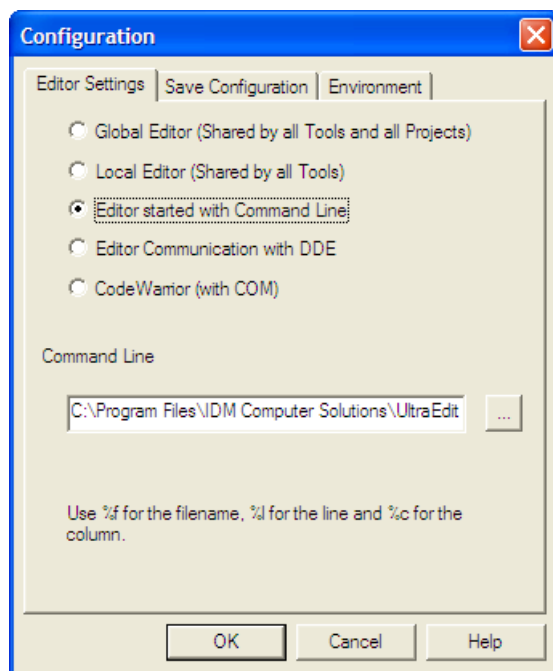


Figure 3-5. Command Line Editor Configuration

### 3.3.3.1 Example of Configuring a Command Line Editor

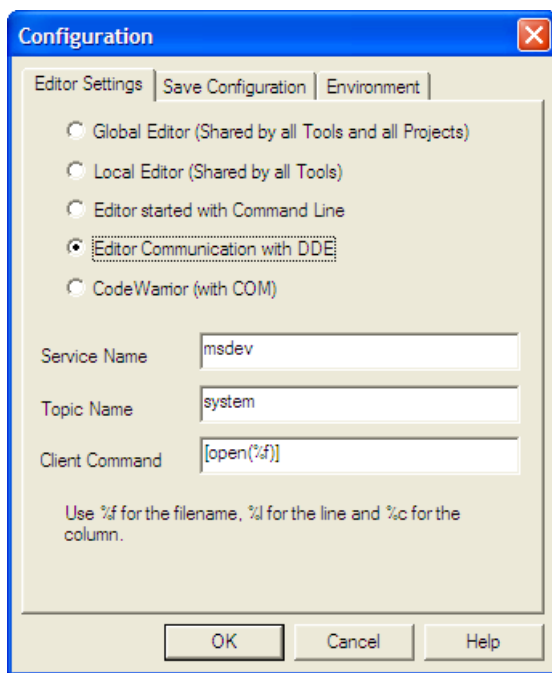
The code listed here portrays the syntax used for configuring an external editors. The following listing can be used for the UltraEdit-32 editor.

#### Listing: UltraEdit-32 configuration

```
C:\UltraEdit32\uedit32.exe %f /#:%l
```

### 3.3.4 Editor Started with DDE

Enter the service, topic and client name to be used for a Dynamic Data Exchange (DDE) connection to the editor (refer to the figure listed below). All entries can have modifiers for the filename and line number, as explained in the [Modifiers](#) section.



**Figure 3-6. DDE Editor Configuration**

For the Microsoft Developer Studio, use the settings in the following listing:

**Listing: Microsoft Developer Studio configuration settings**

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

### 3.3.5 CodeWarrior with COM

If the CodeWarrior with COM is enabled (refer to the figure listed below), the CodeWarrior IDE (registered as a COM server by the installation script) is used as the editor.

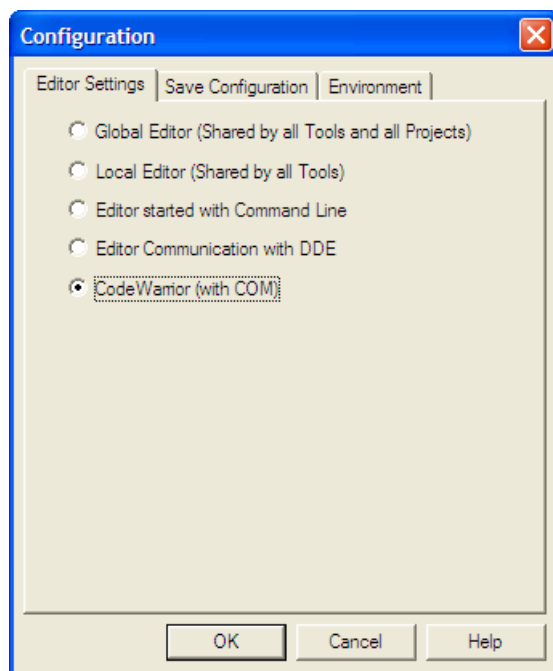


Figure 3-7. COM Editor Configuration

### 3.3.6 Modifiers

The configurations may contain some modifiers to tell the editor which file to open and at which line and column.

- The `%f` modifier refers to the name of the file (including path and extension) where the error 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.

#### CAUTION

The `%l` modifier can only be used with an editor which can be 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 you work with such an editor, you can 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:`

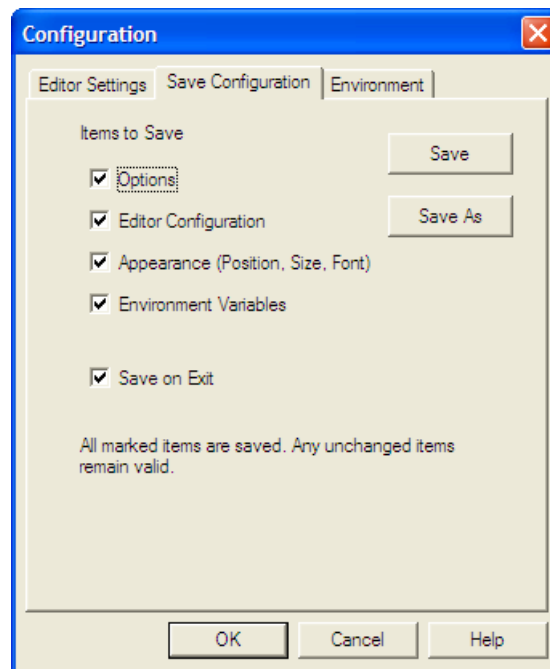
```
\WINAPPS\WINEEDIT\Winedit.exe %f
```

## NOTE

Check your editor manual to define the command line which should be used to start the editor.

### 3.4 Save Configuration Dialog Box

The **Save Configuration** tab of the **Configuration** dialog box contains all options for the save operation. The following image displays the **Save Configuration** tab of the **Configuration** dialog box.



**Figure 3-8. Save Configuration Dialog Box**

In the **Save Configuration** tab, you can select which items to save into a project file when the configuration is saved.

This dialog box has the following configurations:

- **Options** : This item is related to the option and message settings. If this check box is set, the current option and message settings are stored in the project file when the configuration is saved. By disabling this check box, changes done to the option and message settings are not saved, and the previous settings remain valid.
- **Editor Configuration** : This item is related to the editor settings. If you set this check box, the current editor settings are stored in the project file when the

configuration is saved. If you disable this check box, the previous settings remain valid.

- **Appearance** : This item is related to many parts like the window position (only loaded at startup time) and the command line content and history. If you set this check box, these settings are stored in the project file when the current configuration is saved. If you disable this check box, the previous settings remain valid.
- **Environment Variables** : With this set, the environment variable changes done in the Environment property panel are also saved.

### NOTE

By disabling selective options only some parts of a configuration file can be written. For example, when the best Assembler options are found, the save option mark can be removed. Then future save commands will not modify the options any longer.

- **Save on Exit**: If this option is set, the Assembler writes the configuration on exit. The Assembler does not prompt you to confirm this operation. If this option is not set, the assembler does not write the configuration at exit, even if options or other parts of the configuration have changed. No confirmation will appear in any case when closing the assembler.

Almost all settings are stored in the project configuration file. The only exceptions are:

- The recently used configuration list.
- All settings in the Save Configuration dialog box.

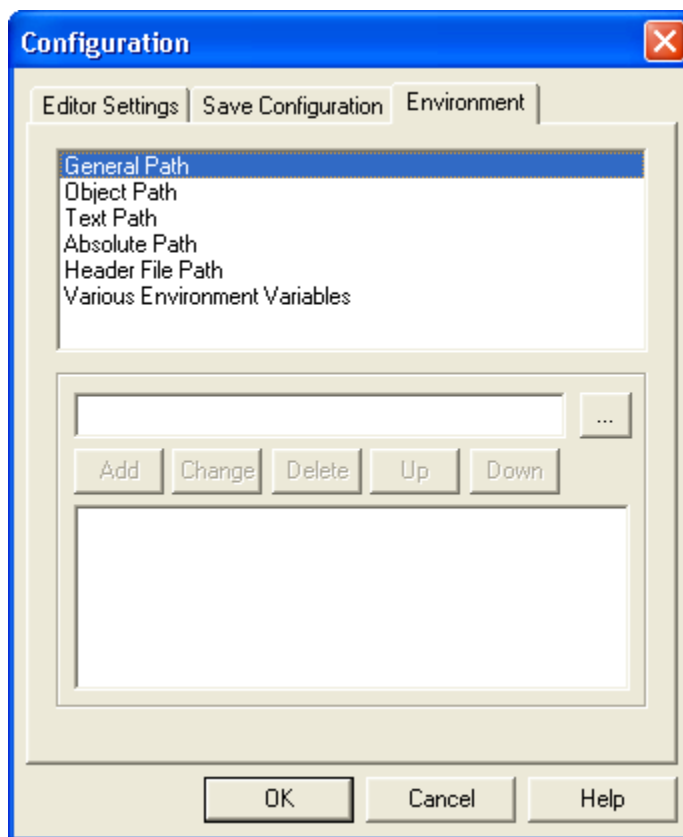
### NOTE

The configurations of the Assembler can, and in fact are intended to, coexist in the same file as the project configuration of other tools and the IDF. When an editor is configured by the shell, the assembler can read this content out of the project file, if present. The default project configuration filename is `project.ini`. The assembler automatically opens an existing `project.ini` in the current directory at startup. Also when using the **-Prod: Specify project file at startup** assembler option at startup or loading the configuration manually, a different name other than `project.ini` can be chosen.



### 3.4.1 Environment Configuration Dialog Box

The **Environment** tab of the **Configuration** dialog box is used to configure the environment. The following image displays the **Environment** tab.



**Figure 3-9. Environment Configuration Dialog Box**

The content of the dialog box is read from the actual project file out of the [ Environment Variables] section.

The following table lists the available variables:

**Table 3-5. Path Environment Variables**

Path	Environment variable
General	GENPATH
Object	OBJPATH
Text	TEXTPATH
Absolute	ABSPATH
Header File	LIBPATH

*Various Environment Variables:* other variables not covered in the above table.

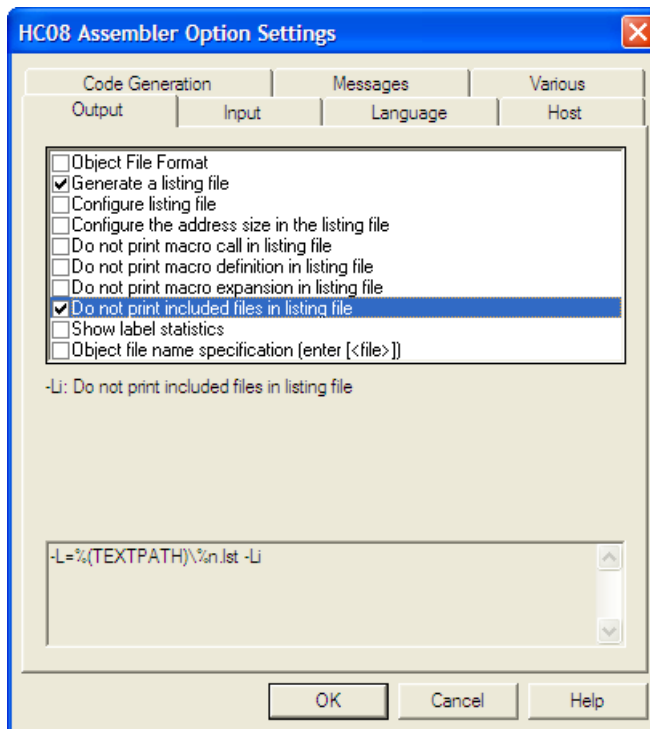
The following buttons are available for the Configuration dialog box:

- **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

Note that the variables are written to the project file only if you press the **Save** button (or using **File -> Save Configuration** or **CTRL-S** ). In addition, it can be specified in the **Save Configuration** dialog box if the environment is written to the project file or not.

### 3.5 Option Settings Dialog Box

Use this dialog box to set or reset assembler options. The following image displays the **HCS08 Assembler Option Settings** dialog box.



**Figure 3-10. Option Settings dialog box**

The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet:

**Table 3-6. Option Settings Options**

Group	Description
Output	Lists options related to the output files generation (which kind of file should be generated).
Input	Lists options related to the input files.
Language	Lists options related to the programming language (ANSI-C, C++, etc.)
Host	Lists options related to the host.
Code Generation	Lists options related to code generation (memory models, etc.)
Messages	Lists options controlling the generation of error messages.
Various	Lists various additional options, such as options used for compatibility.

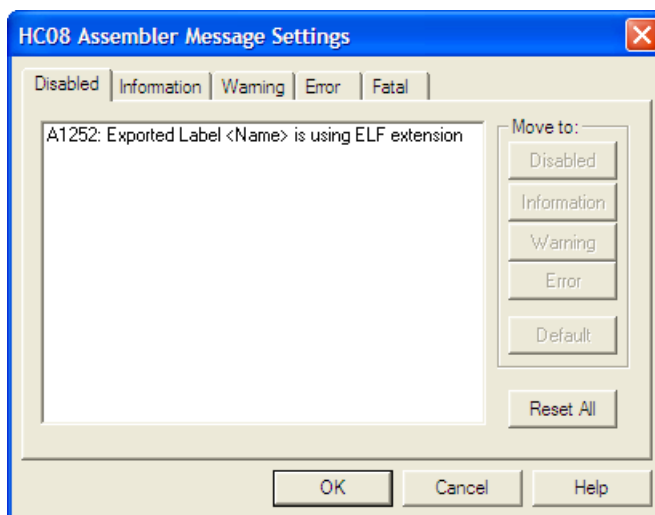
An assembler option is set when the check box in front of it is checked. To obtain more detailed information about a specific option, select it and press the **F1** key or the **Help** button. To select an option, click once on the option text. The option text is then displayed inverted.

When the dialog box is opened and no option is selected, pressing the **F1** key or the **Help** button shows the help about this dialog box.

The available options are listed in the [Assembler Options](#) chapter.

## 3.6 Message Settings Dialog Box

You can use the **Message Settings** (refer to the figure listed below) dialog box to map messages to a different message class.



**Figure 3-11. Message Settings Dialog Box**

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. The following table lists the options available in the **Message Settings** dialog box:

**Table 3-7. Message Settings Options**

Button	Description
Move to: Disabled	Disables selected messages. The disabled messages will no longer be displayed.
Move to: Information	Changes selected messages to information messages.
Move to: Warning	Changes selected messages to warning messages.
Move to: Error	Changes selected messages to error messages.
Move to: Default	Changes selected messages to their default message types.
Reset All	Resets all messages to their default message types.
OK	Exits this dialog box and saves any changes.
Cancel	Exits this dialog box without accepting any changes.
Help	Displays online help about this dialog box.

The following table lists and describes the tabs available in the dialog box for each message group:

**Table 3-8. Message Group**

Message Group	Description
Disabled	Lists all disabled messages. That means that messages displayed in the tab page will not be displayed by the Assembler.
Information	Lists all information messages. Information messages informs about action taken by the Assembler.

*Table continues on the next page...*

**Table 3-8. Message Group (continued)**

Message Group	Description
Warning	Lists all warning messages. When such a message is generated, translation of the input file continues and an object file will be generated.
Error	Lists all error messages. When such a message is generated, translation of the input file continues, but no object file will be generated.
Fatal	Lists all fatal error messages. When such a message is generated, translation of the input file stops immediately. Fatal messages cannot be changed. They are only listed to call context help.

Each message has its own character (^A' for Assembler message) followed by a 4- or 5-digit number. This number allows an easy search for the message on-line help.

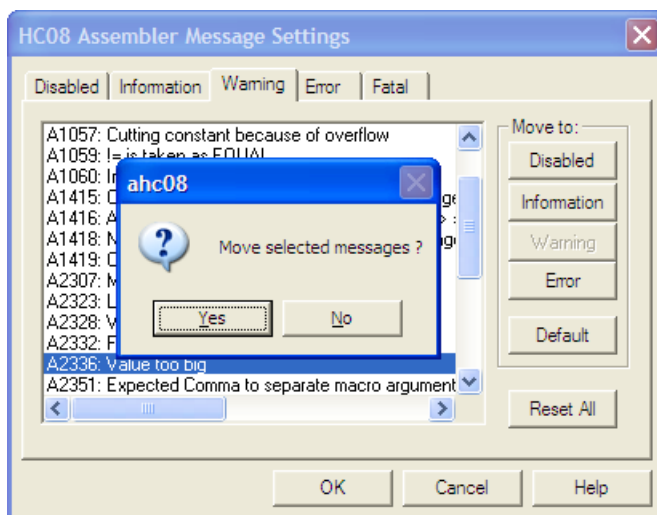
### 3.6.1 Changing the Class Associated with a Message

You can configure your own mapping of messages to the different classes. To do this, 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 dialog box and then click the button associated with the class where you want to move the message.

#### 3.6.1.1 Example

To define the `A2336: Value too big` warning as an error message:

- Click the **Warning** tab to display the list of all warning messages.
- Click on the `A2336: Value too big` string in the to select the message.
- Click **Error** to define this message as an error message. The *<Microcontroller>* dialog box appears, as the following image displays.



**Figure 3-12. Microcontroller Dialog Box**

- Click **Yes** to close the dialog box

**NOTE**

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

**NOTE**

The **Move to** buttons are enabled when all selected messages can be moved. 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 with the **OK** button. If you close it using the **Cancel** button, the previous message mapping remains valid.

### 3.7 About Dialog Box

The **About** dialog box can be opened with the menu **Help > About**. The **About** dialog box contains much information including the current directory and the versions of subparts of the Assembler. The main Assembler version is displayed separately on top of the dialog box.

With the **Extended Information** button it is possible to get license information about all software components in the same directory of the executable.

Click **OK** to close this dialog box.

## NOTE

During assembling, the subversions of the subparts cannot be requested. They are only displayed if the Assembler is not processing files.

## 3.8 Specifying Input File

There are different ways to specify the input file which must be assembled. During assembling of a source file, the options are set according to the configuration performed by the user in the different dialog boxes and according to the options specified on the command line.

Before starting to assemble a file, make sure you have associated a working directory with your assembler.

### 3.8.1 Use Command Line in Toolbar to Assemble

You can use the command line to assemble a new file or to reassemble a previously created file.

### 3.8.2 Assembling a New File

A new filename and additional assembler options can be entered in the command line. The specified file is assembled when you click the **Assemble** button in the toolbar or when you press the enter key.

### 3.8.3 Assembling a File which has Already been Assembled

The commands executed previously can be 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 will be processed when the button **Assemble** in the toolbar is selected.

### 3.8.4 Use File > Assemble Entry

When the menu entry **File > Assemble** is selected, a standard file **Open File** dialog box is opened, displaying the list of all the \*.asm files in the project directory. You can browse to get the name of the file that you want to assemble. Select the desired file and click **Open** in the **Open File** dialog box to assemble the selected file.

### 3.8.5 Use Drag and Drop

A filename can be dragged from an external software (for example the *File Manager/Explorer*) and dropped into the assembler window. The dropped file will be assembled when the mouse button is released in the assembler window. If a file being dragged has the \*.ini extension, it is considered to be a configuration and it is immediately loaded and not assembled. To assemble a source file with the \*.ini extension, use one of the other methods.

## 3.9 Message/Error Feedback

After assembly, there are several ways to check where different errors or warnings have been detected. The default format of the error message is as shown in the following listing.

### Listing: Typical error feedback message

```
Default configuration 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>
```

A typical error message is like the one in the following listing.

### Listing: Error message example

```
>> in "C:\Freescale\demo\fiboerr.asm", line 18, col 0, pos 722
    DC    label
    ^
```



```
ERROR A1104: Undeclared user defined symbol: label
```

For different message formats, see the following Assembler options:

- `-WmsgFi` (`-WmsgFiv`, `-WmsgFim`): Set message file format for interactive mode
- `-WmsgFob`: Message format for batch mode
- `-WmsgFoi`: Message format for interactive mode
- `-WmsgFonf`: Message format for no file information
- `-WmsgFonp`: Message format for no position information.

### 3.9.1 Use Information from Assembler Window

Once a file has been assembled, the Assembler window content area displays the list of all the errors or warnings detected.

The user can use his usual editor to open the source file and correct the errors.

### 3.9.2 Use User-defined Editor

The editor for *Error Feedback* can be configured using the **Configuration** dialog box. Error feedback is performed differently, depending on whether or not the editor can be started with a line number.

### 3.9.3 Line Number can be Specified on the Command Line

Editors like *UltraEdit-32* or *WinEdit* (v95 or higher) can be started with a line number in the command line. When these editors have been correctly configured, they can be started automatically by double clicking on an error message. The configured editor will be started, the file where the error occurs is automatically opened and the cursor is placed on the line where the error was detected.

### 3.9.4 Line Number cannot be Specified on the Command Line

Editors like *WinEditv31* or lower, *Notepad*, or *Wordpad* cannot be started with a line number in the command line. When these editors have been correctly configured, they can be started automatically by double-clicking on an error message. The configured editor will be started, and the file is automatically opened where the error occurs. To scroll to the position where the error was detected, you have to:

1. Switch to the assembler again.
2. Click the line on which the message was generated. This line is highlighted on the screen.
3. Copy the line in the clipboard by pressing **Ctrl + C**.
4. Switch to the editor again.
5. Select **Search > Find** ; the standard **Find** dialog box appears.
6. Paste the contents of the clipboard in the Edit box by pressing **Ctrl + V**.
7. Click **Forward**.

The cursor jump to the position where the error was detected.

## Chapter 4 Environment

This part describes the environment variables used by the Assembler. Some environment variables are also used by other tools (e.g., Linker or Compiler), so consult also the respective documentation.

There are three ways to specify an 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](#) assembler option. This is the recommended method and is also supported by the IDE.
- An optional `default.env` file in the current directory. This file is supported for compatibility reasons with earlier versions. The name of this file may be specified using the [ENVIRONMENT: Environment file specification](#) environment variable. Using the `default.env` file is not recommended.
- Setting environment variables on system level (DOS level). This is also not recommended.

Various parameters of the Assembler may be set in an environment using the environment variables. The syntax is always the same as the following listing shows:

### Listing: Syntax for setting environment variables

```
Parameter: KeyName=ParamDef
```

The following listing shows a typical example of setting an environment variable.

### Listing: Setting the GENPATH environment variable

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

These parameters may be defined in several ways:

- Using system environment variables supported by your operating system.

#### Current directory

- Putting the definitions in a file called `default.env` (`.hidefaults` for UNIX) in the default directory.
- Putting the definitions in a file given by the value of the `ENVIRONMENT` system environment variable.

#### NOTE

The default directory mentioned above can be set via the `DEFAULTDIR` system environment variable.

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

#### NOTE

The environment may also be changed using the [-Env: Set environment variable](#) assembler option.

## 4.1 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` or `.hidefaults`)

Normally, the current directory of a launched tool is determined by the operating system or by the program that launches another one (e.g., IDE, Make Utility, etc.).

For the UNIX operating system, the current directory for an 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 quite complex:

- If the tool is launched using the File Manager/ Explorer, the current directory is the location of the launched executable tool.
- If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon in its properties.
- If the tool is launched by dragging a file on the icon of the executable tool on the desktop, the directory on the desktop is the current directory.

- If the tool is launched by another launching tool with its own current directory specification (e.g., an editor as IDE, a Make utility, etc.), the current directory is the one specified by the launching tool.
- When a local project file is loaded, the current directory is set to the directory which contains the local project file. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for an assembly source file does not change the current directory.

To overwrite this behavior, the [DEFAULTDIR: Default current directory](#) system environment variable may be used.

The current directory is displayed among other information with the [-V: Prints the Assembler version](#) assembler option and in the *About* box.

## 4.2 Environment macros

It is possible to use macros in your environment settings, as the following listing displays.

### Listing: Using a macro for setting environment variables

```
MyVAR=C:\test
TEXTPATH=$(MyVAR)\txt

OBJPATH=${MyVAR}\obj
```

In the above listed example, `TEXTPATH` is expanded to `` C:\test\txt'`, and `OBJPATH` is expanded to `` C:\test\obj'`.

From the example above, you can see that you either can use `$ ()` or `$ {}`. However, the variable referenced has to be defined somewhere.

In addition, the following special variables in [Listing: Usual locations for the mcutools.ini files](#) are allowed. Note that they are case-sensitive and always surrounded by `{}`. Also the variable content contains a directory separator `` \'` as well.

```
{Compiler}
```

This is the path of the directory one level higher than the directory for executable tool. That is, if the executable is `C:\Freescale\prog\linker.exe`, then the variable is `C:\Freescale\`. Note that `{Compiler}` is also used for the Assembler.

```
{Project}
```

Path of the directory containing the current project file. For example, if the current project file is `C:\demo\project.ini`, the variable contains `C:\demo\`.

```
{System}
```

This is the path where Windows OS is installed, e.g., `C:\WINNT\`.

## 4.3 Global initialization file - mctools.ini (PC only)

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

The following listing shows two typical locations used for the `mctools.ini` files.

### Listing: Usual locations for the mctools.ini files

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

If a tool is started in the `D:\INSTALL\prog\` directory, the initialization file located in the same directory as the tool is used (`D:\INSTALL\prog\mctools.ini`).

But if the tool is started outside of the `D:\INSTALL\prog` directory, the initialization file in the *Windows* directory is used (`C:\WINDOWS\mctools.ini`).

## 4.4 Local configuration file (usually project.ini)

The Assembler does not change the `default.env` file in any way. The Assembler only reads the contents. All the configuration properties are stored in the configuration file. The same configuration file can and is intended to be used by different applications.

The processor name is encoded into the section name, so that the Assembler for different processors can use the same file without any overlapping. Different versions of the same Assembler are using the same entries. This usually only leads to a potential problem when options only available in one version are stored in the configuration file. In such situations, two files must be maintained for the different Assembler versions. If no incompatible options are enabled when the file is last saved, the same file can be used for both Assembler versions.

The current directory is always the directory that holds the configuration file. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the whole `default.env` file is also reloaded.

When a configuration file is loaded or stored, the options located in the **ASMOPTIONS: Default assembler options** environment variable are reloaded and added to the project's options.

This behavior has to be noticed when in different directories different `default.env` files exist which contain incompatible options in their `ASMOPTIONS` environment variables. When a project is loaded using the first `default.env` file, its `ASMOPTIONS` options are added to the configuration file. If this configuration is then stored in a different directory, where a `default.env` file exists with these incompatible options, the Assembler adds the options and remarks the inconsistency. Then a message box appears to inform the user that those options from the `default.env` file were not added. In such a situation, the user can either remove the options from the configuration file with the advanced option dialog box or he can remove the option from the `default.env` file with the shell or a text editor depending upon which options should be used in the future.

At startup, the configuration stored in the `project.ini` file located in the current Paths Local Configuration File Entries documents the sections and entries you can put in a `project.ini` file.

Most environment variables contain path lists telling where to look for files. A path list is a list of directory names separated by semicolons following the syntax, as listed in the following listing:

### **Listing: Syntax used for setting path lists of environment variables**

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

The following listing shows a typical example of setting an environment variable.

### **Listing: Setting the paths for the GENPATH environment variable**

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

If a directory name is preceded by an asterisk (\*), the programs recursively search that whole directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list. The following listing shows the use of an asterisk (\*) for recursively searching the entire C drive for a configuration file with a `\INSTALL\LIB` path.

### **Listing: Recursive search for a continuation line**

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

**NOTE**

Some DOS/UNIX environment variables (like `GENPATH`, `LIBPATH`, etc.) are used. For further details refer to the section [Environment variables details](#).

We strongly recommend working with the Shell and setting the environment by means of a `default.env` file in your project directory. (This `project dir` can be set in the Shell's 'Configure' dialog box). Doing it this way, you can have different projects in different directories, each with its own environment.

**NOTE**

When starting the Assembler from an external editor, do *not* set the `DEFAULTDIR` system environment variable. If you do so and this variable does not contain the project directory given in the editor's project configuration, files might not be put where you expect them to be put!

A synonym also exists for some environment variables. Those synonyms may be used for older releases of the Assembler, but they are deprecated and thus they will be removed in the future.

## 4.5 Line continuation

It is possible to specify an environment variable in an environment file (`default.env` or `.hidefaults`) over multiple lines using the line continuation character ``\``. The following listing shows using multiple lines for an environment variable.

**Listing: Using multiple lines for an environment variable**

```
ASMOPTIONS=\
    -W2\
    -WmsgNe=10
```

The above listing is the same as the alternate source code in the following listing.

**Listing: Alternate form of using multiple lines**

```
ASMOPTIONS=-W2 -WmsgNe=10
```

But this feature may be dangerous when used together with paths, as following listing shows:



### Listing: A path is included by the line continuation character

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

will result in

GENPATH=.TEXTFILE=.\txt
```

To avoid such problems, we recommend that you use a semicolon (;) at the end of a path if there is a backslash (\) at the end, as the following listing shows:

### Listing: Recommended style whenever a backslash is present

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

## 4.6 Environment variables details

The remainder of this section is devoted to describing each of the environment variables available for the Assembler. The environment variables are listed in alphabetical order and each is divided into several sections. The following table lists and describes the environmental variables.

**Table 4-1. Topics used for describing environment variables**

Topic	Description
Tools	Lists tools which are using this variable.
Synonym (where one exists)	A synonym exists for some environment variables. These synonyms may be used for older releases of the Assembler but they are deprecated and they 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 (if one exists)	Shows the default setting for the variable if one exists.
Description	Provides a detailed description of the option and its usage.
Example	Gives an example of usage and effects of the variable where possible. An example shows an entry in the <code>default.env</code> for the PC or in the <code>.hidefaults</code> for UNIX.
See also (if needed)	Names related sections.

## 4.6.1 ABSPATH: Absolute file path

### Tools

Compiler, Assembler, Linker, Decoder, or Debugger

### Syntax

```
ABSPATH={<path>}
```

### Arguments

<path>: Paths separated by semicolons, without spaces

### Description

This environment variable is only relevant when absolute files are directly generated by the Macro Assembler instead of relocatable object files. When this environment variable is defined, the Assembler will store the absolute files it produces in the first directory specified there. If `ABSPATH` is not set, the generated absolute files will be stored in the directory where the source file was found.

### Example

```
ABSPATH=\sources\bin;..\..\headers;\usr\local\bin
```

## 4.6.2 ASMOPTIONS: Default assembler options

### Tools

Assembler

### Syntax

```
ASMOPTIONS={<option>}
```

### Arguments

<option>: Assembler command-line option

### Description

If this environment variable is set, the Assembler appends its contents to its command line each time a file is assembled. It can be used to globally specify certain options that should always be set, so you do not have to specify them each time a file is assembled.

Options enumerated there must be valid assembler options and are separated by space characters.

### Example

```
ASMOPTIONS=-W2 -L
```

### See also

[Assembler Options](#) chapter

## 4.6.3 COPYRIGHT: Copyright entry in object file

### Tools

Compiler, Assembler, Linker, or Librarian

### Syntax

```
COPYRIGHT=<copyright>
```

### Arguments

<copyright>: copyright entry

### Description

Each object file contains an entry for a copyright string. This information may be retrieved from the object files using the Decoder.

### Example

```
COPYRIGHT=Copyright
```

### See also

- [USERNAME: User Name in object file](#)
- [INCLUDETIME: Creation time in the object file](#)

## 4.6.4 DEFAULTDIR: Default current directory

### Tools

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

### Syntax

```
DEFAULTDIR=<directory>
```

### Arguments

<directory>: Directory to be the default current directory

### Description

The default directory for all tools may be specified with this environment variable. Each of the tools indicated above will take the directory specified as its current directory instead of the one defined by the operating system or launching tool (e.g., editor).

### NOTE

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

### Example

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

### See also

#### [Current directory](#)

"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 tool). 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`)."

## 4.6.5 ENVIRONMENT: Environment file specification

## Tools

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

## Synonym

HIENVIRONMENT

## Syntax

ENVIRONMENT=<file>

## Arguments

<file>: filename with path specification, without spaces

## Description

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

### NOTE

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

## Example

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

## 4.6.6 ERRORFILE: Filename specification error

### Tools

Compiler, Assembler, or Linker

### Syntax

ERRORFILE=<filename>

### Arguments

<filename>: Filename with possible format specifiers

## Default

EDOUT

## Description

The `ERRORFILE` environment variable specifies the name for the error file (used by the Compiler or Assembler).

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').

In case of an improper error filename, a notification box is shown.

## Examples

The following listing lists all errors into the `MyErrors.err` file in the current directory.

### Listing: Naming an error file

```
ERRORFILE=MyErrors.err
```

The following listing lists all errors into the `errors` file in the `\tmp` directory.

### Listing: Naming an error file in a specific directory

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

The following listing lists all errors into a file with the same name as the source file, but with extension `*.err`, into the same directory as the source file, e.g., if we compile a file `\sources\test.c`, an error list file `\sources\test.err` will be generated.

### Listing: Naming an error file as source filename

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

For a `test.c` source file, a `\dir1\test.err` error list file will be generated, as the following listing shows:

### Listing: Naming an error file as source filename in a specific directory

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

For a `\dir1\dir2\test.c` source file, a `\dir1\dir2\errors.txt` error list file will be generated, as the following listed shows:

### Listing: Naming an error file as a source filename with full path

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

If the `ERRORFILE` environment variable is not set, errors are written to the default error file. The default error filename depends on the way the Assembler is started.

If a filename is provided on the assembler command line, the errors are written to the `EDOUT` file in the project directory.

If no filename is provided on the assembler command line, the errors are written to the `err.txt` file in the project directory.

The following listed shows another example for the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called `EDOUT`:

### Listing: Configuring error feedback with WinEdit

```
Installation directory: E:\INSTALL\prog
Project sources: D:\SRC

Common Sources for projects: E:\CLIB

Entry in default.env (D:\SRC\default.env):

ERRORFILE=E:\INSTALL\prog\EDOUT

Entry in WinEdit.ini (in Windows directory):

OUTPUT=E:\INSTALL\prog\EDOUT
```

#### NOTE

You must set this variable if the WinEdit Editor is used, otherwise the editor cannot find the `EDOUT` file.

## 4.6.7 GENPATH: Search path for input file

### Tools

Compiler, Assembler, Linker, Decoder, or Debugger

### Synonym

HIPATH

## Syntax

```
GENPATH={ <path> }
```

## Arguments

<path>: Paths separated by semicolons, without spaces.

## Description

The Macro Assembler will look for the sources and included files first in the project directory, then in the directories listed in the `GENPATH` environment variable.

### NOTE

If a directory specification in this environment variables starts with an asterisk ( \* ), the whole directory tree is searched recursive depth first, i.e., all subdirectories and *their* subdirectories and so on are searched. Within one level in the tree, the search order of the subdirectories is indeterminate.

## Example

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

## 4.6.8 INCLUDETIME: Creation time in the object file

### Tools

Compiler, Assembler, Linker, or Librarian

### Syntax

```
INCLUDETIME= (ON | OFF)
```

### Arguments

ON: Include time information into the object file.



`OFF`: Do not include time information into the object file.

## Default

ON

-

## Description

Normally each object file created contains a time stamp indicating the creation time and data as strings. So 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 SQA 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 because the time stamps are not the same. To avoid such problems this variable may be set 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 may be retrieved from the object files using the Decoder.

## Example

```
INCLUDETIME=OFF
```

## See also

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

## 4.6.9 OBJPATH: Object file path

### Tools

Compiler, Assembler, Linker, or Decoder

### Syntax

```
OBJPATH={ <path> }
```

## Arguments

<path>: Paths separated by semicolons, without spaces

## Description

This environment variable is only relevant when object files are generated by the Macro Assembler. When this environment variable is defined, the Assembler will store the object files it produces in the first directory specified in `path`. If `OBJPATH` is not set, the generated object files will be stored in the directory the source file was found.

## Example

```
OBJPATH=\sources\bin;..\..\headers;\usr\local\bin
```

## 4.6.10 SRECORD: S-Record type

### Tools

Assembler, Linker, or Burner

### Syntax

```
SRECORD=<RecordType>
```

### Arguments

<RecordType>: Forces the type for the S-Record File which must be generated. This parameter may take the value ``s1'`, ``s2'`, or ``s3'`.

### Description

This environment variable is only relevant when absolute files are directly generated by the Macro Assembler instead of object files. When this environment variable is defined, the Assembler will generate an S-Record File containing records from the specified type ( `s1` records when `s1` is specified, `s2` records when `s2` is specified, and `s3` records when `s3` is specified).

### NOTE

If the `SRECORD` environment variable is set, it is the user's responsibility to specify the appropriate type of S-Record File. If you specify `s1` while your code is loaded above `0xFFFF`, the S-

Record File generated will not be correct because the addresses will all be truncated to 2-byte values.

When this variable is not set, the type of S-Record File generated will depend on the size of the address, which must be loaded there. If the address can be coded on 2 bytes, an `s1` record is generated. If the address is coded on 3 bytes, an `s2` record is generated. Otherwise, an `s3` record is generated.

### Example

```
SRECORD=S2
```

## 4.6.11 TEXTPATH: Text file path

### Tools

Compiler, Assembler, Linker, or Decoder

### Syntax

```
TEXTPATH={<path>}
```

### Arguments

`<path>`: Paths separated by semicolons, without spaces.

### Description

When this environment variable is defined, the Assembler will store the listing files it produces in the first directory specified in `path`. If `TEXTPATH` is not set, the generated listing files will be stored in the directory the source file was found.

### Example

```
TEXTPATH=\sources\txt;..\..\headers;\usr\local\txt
```

## 4.6.12 TMP: Temporary directory

## Tools

Compiler, Assembler, Linker, Debugger, or Librarian

## Syntax

```
TMP=<directory>
```

## Arguments

<directory>: Directory to be used for temporary files

## Description

If a temporary file has to be created, normally 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 an error message *Cannot create temporary file*.

### NOTE

`TMP` is an environment variable on the system level (global environment variable). It *CANNOT* be specified in a default environment file (default `.env` or `.hidefaults`).

## Example

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

## See also

[Current directory](#) section

## 4.6.13 USERNAME: User Name in object file

## Tools

Compiler, Assembler, Linker, or Librarian

## Syntax

```
USERNAME=<user>
```

## Arguments

<user>: Name of user

## Description

Each object file contains an entry identifying the user who created the object file. This information may be retrieved from the object files using the decoder.

## Example

```
USERNAME=PowerUser
```

## See also

- [COPYRIGHT](#): Copyright entry in object file
- [INCLUDETIME](#): Creation time in the object file



# Chapter 5

## Files

This chapter covers these topics:

- [Input files](#)
- [Output files](#)
- [File processing](#)

### 5.1 Input files

Input files to the Assembler:

- [Source files](#)
- [Include files](#)

#### 5.1.1 Source files

The Macro Assembler takes any file as input. It does not require the filename to have a special extension. However, we suggest that all your source filenames have the `*.asm` extension and all included files have the `*.inc` extension. Source files will be searched first in the project directory and then in the directories enumerated in [GENPATH: Search path for input file](#).

#### 5.1.2 Include files

The search for include files is governed by the `GENPATH` environment variable. Include files are searched for first in the project directory, then in the directories given in the `GENPATH` environment variable. The project directory is set via the Shell, the Program Manager, or the [DEFAULTDIR: Default current directory](#) environment variable.

## 5.2 Output files

Output files from the Assembler:

- [Object files](#)
- [Absolute files](#)
- [S-Record Files](#)
- [Listing files](#)
- [Debug listing files](#)
- [Error listing file](#)

### 5.2.1 Object files

After a successful assembling session, the Macro Assembler generates an object file containing the target code as well as some debugging information. This file is written to the directory given in the [OBJPATH: Object file path](#) environment variable. If that variable contains more than one path, the object file is written in the first directory given; if this variable is not set at all, the object file is written in the directory the source file was found. Object files always get the `*.o` extension.

### 5.2.2 Absolute files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate directly an absolute file instead of an object file. This file is written to the directory given in the [ABSPATH: Absolute file path](#) environment variable. If that variable contains more than one path, the absolute file is written in the first directory given; if this variable is not set at all, the absolute file is written in the directory the source file was found. Absolute files always get the `*.abs` extension.



### 5.2.3 S-Record Files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate directly an ELF absolute file instead of an object file. In that case an S-Record File is generated at the same time. This file can be burnt into an EPROM. It contains information stored in all the `READ_ONLY` sections in the application. The extension for the generated S-Record File depends on the setting from the [SRECORD: S-Record type](#) environment variable.

- If `SRECORD = S1`, the S-Record File gets the `*.s1` extension.
- If `SRECORD = S2`, the S-Record File gets the `*.s2` extension.
- If `SRECORD = S3`, the S-Record File gets the `*.s3` extension.
- If `SRECORD` is not set, the S-Record File gets the `*.sx` extension.

This file is written to the directory given in the `ABSPATH` environment variable. If that variable contains more than one path, the S-Record File is written in the first directory given; if this variable is not set at all, the S-Record File is written in the directory the source file was found.

### 5.2.4 Listing files

After successful assembling session, the Macro Assembler generates a listing file containing each assembly instruction with their associated hexadecimal code. This file is always generated when the `-L: Generate a listing file` assembler option is activated (even when the Macro Assembler generates directly an absolute file). This file is written to the directory given in the [TEXTPATH: Text file path](#) environment variable. If that variable contains more than one path, the listing file is written in the first directory given; if this variable is not set at all, the listing file is written in the directory the source file was found. Listing files always get the `*.lst` extension. The format of the listing file is described in the [Assembler Listing File](#) chapter.

### 5.2.5 Debug listing files

After successful assembling session, the Macro Assembler generates a debug listing file, which will be used to debug the application. This file is always generated, even when the Macro Assembler directly generates an absolute file. The debug listing file is a duplicate from the source, where all the macros are expanded and the include files merged. This file is written to the directory given in the **OBJPATH: Object file path** environment variable. If that variable contains more than one path, the debug listing file is written in the first directory given; if this variable is not set at all, the debug listing file is written in the directory the source file was found. Debug listing files always get the \*.dbg extension.

## 5.2.6 Error listing file

If the Macro Assembler detects any errors, it does not create an object file but does create an error listing file. This file is generated in the directory the source file was found (see **ERRORFILE: Filename specification error**).

If the Assembler's window is open, it displays the full path of all include files read. After successful assembling, the number of code bytes generated is displayed, too. In case of an error, the position and filename where the error occurs is displayed in the assembler window.

If the Assembler is started from the *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 Microsoft default format in a file called `EDOUT`. Use *WinEdit's Next Error* or CodeWright's *Find Next Error* command to see both error positions and the error messages.

### 5.2.6.1 Interactive mode (Assembler window open)

If `ERRORFILE` is set, the Assembler 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.

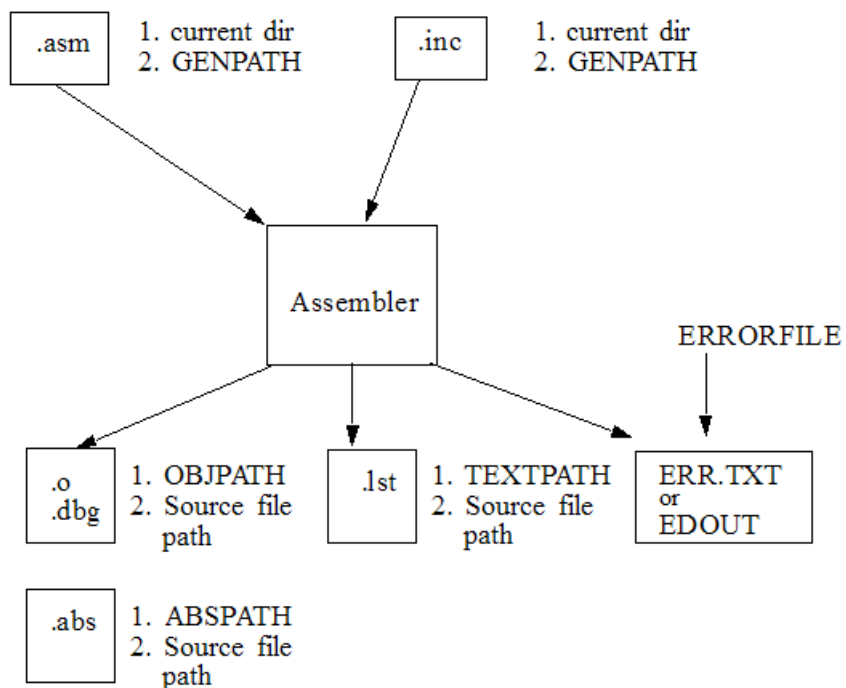
### 5.2.6.2 Batch mode (Assembler window not open)

If `ERRORFILE` is set, the Assembler 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.

### 5.3 File processing

The following image shows the priority levels for the various files used by the Assembler.



**Figure 5-1. Files used with the Assembler**



# Chapter 6

## Assembler Options

This chapter describes the assembler options for HCS08 and RS08 architectures.

### 6.1 Assembler Options

This chapter describes the assembler options for HCS08 and RS08 architectures.

### 6.2 Assembler Option details

The remainder of this section is devoted to describing each of the assembler options available for the Assembler. The options are listed in alphabetical order and each is divided into several sections. The following table lists and describes the topics listed in the assembler options.

**Table 6-1. Assembler option details**

Topic	Description
Group	Output, Input, Language, Host, Code Generation, Messages, or Various.
Scope	Application, Assembly 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.
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. Assembler settings, source code and/or Linker PRM files are displayed where applicable. The examples shows an

*Table continues on the next page...*

**Table 6-1. Assembler option details (continued)**

Topic	Description
	entry in the <code>default.env</code> for the PC or in the <code>.hidefaults</code> for UNIX.
See also (if needed)	Names related options.

## 6.2.1 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.

The following table lists and describes the supported modifiers.

**Table 6-2. Special modifiers for assembler options**

Modifier	Description
<code>%p</code>	Path including file separator
<code>%N</code>	Filename in strict 8.3 format
<code>%n</code>	Filename without its extension
<code>%E</code>	Extension in strict 8.3 format
<code>%e</code>	Extension
<code>%f</code>	Path + filename without its 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 `%'

### 6.2.1.1 Examples using special modifiers

The assumed path and filename (filename base for the modifiers) used for the examples are displayed in the following listing.

**Listing: Example filename and path used for the following examples**

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

Using the `%p` modifier as the following listing displays the path with a file separator but without the filename.

**Listing: %p gives the path only with the final file separator**

```
C:\Freescale\my demo\
```

Using the %N modifier only displays the filename in 8.3 format but without the file extension, as the following listing displays.

**Listing: %N results in the filename in 8.3 format (only the first 8 characters)**

```
TheWhole
```

The %n modifier returns the entire filename but with no file extension, as the following listing displays.

**Listing: %n returns just the filename without the file extension**

```
TheWholeThing
```

Using %E as a modifier returns the first three characters in the file extension, as the following listing displays.

**Listing: %E gives the file extension in 8.3 format (only the first 3 characters)**

```
myE
```

If you want the entire file extension, use the %e modifier, as the following listing displays.

**Listing: %e is used for returning the whole extension**

```
myExt
```

The %f modifier returns the path and the filename without the file extension, as the following listing displays.

**Listing: %f gives the path plus the filename (no file extension)**

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

The path in *Listing: Example filename and path used for the following examples* contains a space, therefore using %" or %' is recommended, as the following listings displays.

**Listing: Use %" %f%" in case there is a space in its path, filename, or extension**

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

**Listing: Use %' %f%' where there is a space in its path, filename, or extension**

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

Using %(envVariable) an environment variable may be used. A file separator following %(envVariable) is ignored if the environment variable is empty or does not exist. If TEXTPATH is set as in following listing, then \$(TEXTPATH)\myfile.txt is expressed as in next listing.

## Listing: Example for setting TEXTPATH

```
TEXTPATH=C:\Freescale\txt
```

## Listing: \$(TEXTPATH)\myfile.txt where TEXTPATH is defined

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

However, if `TEXTPATH` does not exist or is empty, then `$(TEXTPATH)\myfile.txt` is expressed as in the following listing.

## Listing: \$(TEXTPATH)\myfile.txt where TEXTPATH does not exist

```
myfile.txt
```

It is also possible to display the percent sign by using `%%`. `%e%` allows the expression of a percent sign after the extension as in the following listing.

## Listing: %% allows a percent sign to be expressed

```
myExt%
```

## 6.3 List of Assembler Option

The following listed are the command line option you can use with the Assembler.

### NOTE

Not all tools options have been defined for this release. All descriptions will be available in an upcoming release.

**Table 6-3. Assembler Options**

Assembler option
-ArgFile: Specify a file from which additional command line options will be read
-AsmDbg: Emit assembly source file information in debug sections
-Ci: Switch case sensitivity on label names OFF
-CMacAngBrack: Angle brackets for grouping Macro Arguments
-CMacBrackets: Square brackets for macro arguments grouping
-Compat: Compatibility modes
-CS08/-C08/-CRS08: Derivative family
-D: Define Label
-DefLabel: Improves support for data allocation directives
-Env: Set environment variable
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format

*Table continues on the next page...*



**Table 6-3. Assembler Options (continued)**

Assembler option
-H: Short Help
-I: Include file path
-L: Generate a listing file
-Lasmc: Configure listing file
-Lasms: Configure the address size in the listing file
-Lc: No Macro call in listing file
-Ld: No macro definition in listing file
-Le: No Macro expansion in listing file
-Li: No included file in listing file
-Lic: License information
-LicA: License information about every feature in directory
-LicBorrow: Borrow license feature
-LicWait: Wait until floating license is available from floating License Server
-LI: Show label statistics
-Lrefs: Emit Cross References list of symbols
-M (-Ms, -Mt): Memory model
-MacroNest: Configure maximum macro nesting
-MCUasm: Switch compatibility with MCUasm ON
-MMU: Enable Memory Management Unit (MMU) Support
-N: Display notify box
-NoBeep: No beep in case of an error
-NoDebugInfo: No debug information for ELF/DWARF files
-NoEnv: Do not use environment
-ObjN: Object filename specification
-Prod: Specify project file at startup
-Struct: Support for structured types
-V: Prints the Assembler version
-View: Application standard occurrence
-W1: No information messages
-W2: No information and warning messages
-WErrFile: Create "err.log" error file
-Wmsg8x3: Cut filenames in Microsoft format to 8.3
-WmsgCE: RGB color for error messages
-WmsgCF: RGB color for fatal messages
-WmsgCI: RGB color for information messages
-WmsgCU: RGB color for user messages
-WmsgCW: RGB color for warning messages
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode

*Table continues on the next page...*

**Table 6-3. Assembler Options (continued)**

Assembler option
-WmsgFob: Message format for batch mode
-WmsgFoi: Message format for interactive mode
-WmsgFonf: Message format for no file information
-WmsgFonp: Message format for no position information
-WmsgNe: Number of error messages
-WmsgNi: Number of Information messages
-WmsgNu: Disable user messages
-WmsgNw: Number of Warning messages
-WmsgSd: Setting a message to disable
-WmsgSe: Setting a message to Error
-WmsgSi: Setting a message to Information
-WmsgSw: Setting a Message to Warning
-WOutFile: Create error listing file
-WStdout: Write to standard output

### 6.3.1 -ArgFile: Specify a file from which additional command line options will be read

#### Group

HOST

#### Scope

Function

#### Syntax

`-ArgFile<filename>`

#### Arguments

`<filename>`: Specify filename that has options to be passed to command line

#### Description

The options present in file are appended to existing command line options.

#### Example

```
option.txt  
-M  
Linker.exe -ArgFileoption.txt test.prm
```

This is equivalent to `linker.exe -M test.prm` and linker generates output file `test.map`

### 6.3.2 -AsmDbg: Emit assembly source file information in debug sections

#### Group

CODE GENERATION

#### Scope

Function

#### Syntax

```
-AsmDbg
```

#### Arguments

None

#### Description

This option when enabled, passes the assembly source file name information to DWARF sections. When the output `.abs` file is debugged, the actual assembly source file is displayed instead of intermediary `<filename>.dbg` file.

### 6.3.3 -Ci: Switch case sensitivity on label names OFF

#### Group

Input

#### Scope

Assembly Unit

## Syntax

-Ci

## Arguments

None

## Default

None

## Description

This option turns off case sensitivity on label names. When this option is activated, the Assembler ignores case sensitivity for label names. If the Assembler generates object files but not absolute files directly ( `-FA2` assembler option), the case of exported or imported labels must still match. Or, the `-Ci` assembler option should be specified in the linker as well.

## Example

When case sensitivity on label names is switched off, the Assembler will not generate an error message for the assembly source code in the following listing.

### Listing: Example assembly source code

```
        ORG $200
entry:  NOP

        BRA Entry
```

The instruction `BRA Entry` branches on the `entry` label. The default setting for case sensitivity is `ON`, which means that the Assembler interprets the labels `Entry` and `entry` as two distinct labels.

## See also

[-F \(-Fh, -F2o, -FA2o, -F2, -FA2\): Output file format](#) assembler option

## 6.3.4 -CMacAngBrack: Angle brackets for grouping Macro Arguments

## Group

Language

**Scope**

Application

**Syntax**

-CMacAngBrack (ON|OFF)

**Arguments**

ON or  
OFF

**Default**

None

**Description**

This option controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the [? ?] syntax instead.

**See also**

[Macro argument grouping](#)

[-CMacBrackets: Square brackets for macro arguments grouping option](#)

### 6.3.5 -CMacBrackets: Square brackets for macro arguments grouping

**Group**

Language

**Scope**

Application

**Syntax**

## List of Assembler Option

-CMacBrackets (ON|OFF)

### Arguments

ON or  
OFF

### Default

ON

### Description

This option controls the availability of the [? ?] syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for [? in the macro invocation context.

### See also

[Macro argument grouping](#)

[-CMacAngBrack: Angle brackets for grouping Macro Arguments option](#)

## 6.3.6 -Compat: Compatibility modes

### Group

Language

### Scope

Application

### Syntax

-Compat [= { ! | = | c | s | f | \$ | a | b }

### Arguments

See below.

### Default

None

## Description

This option controls some compatibility enhancements of the Assembler. The goal is not to provide 100% compatibility with any other Assembler but to make it possible to reuse as much as possible. The various suboptions control different parts of the assembly:

- `! =`: Operator `!=` means equal

The Assembler takes the default value of the `!=` operator as *not equal*, as it is in the C language. For compatibility, this behavior can be changed to *equal* with this option. Because the danger of this option for existing code, a message is issued for every `!=` which is treated as equal.

- `! :` Support additional `!` operators

The following additional operators are defined when this option is used:

- `! ^`: exponentiation
- `! m`: modulo
- `! @`: signed greater or equal
- `! g`: signed greater
- `! %`: signed less or equal
- `! t`: signed less than
- `! $`: unsigned greater or equal
- `! s`: unsigned greater
- `! &`: unsigned less or equal
- `! l`: unsigned less
- `! n`: one complement
- `! w`: low operator
- `! h`: high operator

The *default* values for the following `!` operators are defined:

`! .`: binary AND

`! x`: exclusive OR

`! +`: binary OR

- `! c`: Alternate comment rules

With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler does issue a warning if such a comment does not start with a "\*" or a ";".

## Examples

The following listing demonstrates that when `-Compat=c`, comments can start with a `*`.

### Listing: Comments starting with an asterisk (\*)

```
NOP * Anything following an asterisk is a comment.
```

When the `-Compat=c` assembler option is used, the first `DC.B` directive in the following listing, has `" + 1 , 1"` as a comment. A warning is issued because the comment does not start with a `;"` or a `"*"`. With `-Compat=c`, this code generates a warning and three bytes with constant values 1, 2, and 1. Without it, this code generates four 8-bit constants of 2, 1, 2, and 1.

### Listing: Implicit comment start after a space

```
DC.B 1 + 1 , 1
DC.B 1+1,1
```

- **s:** Symbol prefixes

With this suboption, some compatibility prefixes for symbols are supported. With this option, the Assembler accepts `"pgz:"` and `"byte:"` prefixed for symbols in `XDEFs` and `XREFs`. They correspond to `XREF.B` or `XDEF.B` with the same symbols without the prefix.

- **f:** Ignore `FF` character at line start

With this suboption, an otherwise improper character recognized from feed character is ignored.

- **§:** Support the `$` character in symbols

With this suboption, the Assembler supports to start identifiers with a `§` sign.

- **a:** Add some additional directives

With this suboption, some additional directives are added for enhanced compatibility.



The Assembler actually supports a `SECT` directive as an alias of the usual [SECTION - Declare Relocatable Section](#) assembly directive. The `SECT` directive takes the section name as its first argument.

- `b`: support the `FOR` directive

With this suboption, the Assembler supports a [FOR - Repeat assembly block](#) assembly directive to generate repeated patterns more easily without having to use recursive macros.

### 6.3.7 -CS08/-C08/-CRS08: Derivative family

#### Group

Code Generation

#### Scope

Application

#### Syntax

```
-C08 | -CS08 | -CRS08
```

#### Arguments

None

#### Default

```
-C08
```

#### Description

The Assembler supports three different HC08-derived cores. The HC08 itself ( `-C08` ), the enhanced HCS08 ( `-CS08` ), and the RS08 ( `-CRS08` ).

The HCS08 family supports additional addressing modes for the `CPHX`, `LDHX`, and `STHX` instructions and also a new `BGND` instruction. All these enhancements are allowed when the `-CS08` option is specified. All instructions and addressing modes available for the HC08 are also available for the HCS08 so that this core remains binary compatible with its predecessor.

The RS08 family does not support all instructions and addressing modes of the HC08. Also, the encoding of the supported instructions is not binary compatible.

**Table 6-4. Table of new instructions or addressing modes for the HCS08**

Instruction	Addr. mode	Description
LDHX	EXT IX IX1 IX2 SP1	load from a 16-bit absolute address load HX via 0,X load HX via 1,X...255,X load HX via old HX+ any offset load HX from stack
STHX	EXT SP1	store HX to a 16-bit absolute address store HX to stack
CPHX	EXT SP1	compare HX with a 16-bit address compare HX with the stack
BGND		enter the Background Debug Mode

## 6.3.8 -D: Define Label

### Group

Input

Scope

Assembly Unit

### Syntax

```
-D<LabelName> [=<Value>]
```

### Arguments

<LabelName>: Name of label.

<Value>: Value for label. 0 if not present.

### Default

0 for value.

### Description

This option behaves as if a `Label: EQU Value` is at the start of the main source file. When no explicit value is given, 0 is used as the default.

This option can be used to build different versions with one common source file.

## Example

Conditional inclusion of a copyright notice. See the following listings.

### Listing: Source code that conditionally includes a copyright notice

```

YearAsString: MACRO
    DC.B $30+(\1 /1000)%10

    DC.B $30+(\1 / 100)%10

    DC.B $30+(\1 / 10)%10

    DC.B $30+(\1 / 1)%10

ENDM

ifdef ADD_COPYRIGHT

    ORG $1000

    DC.B "Copyright by "

    DC.B "John Doe"

ifdef YEAR

    DC.B " 1999-"

    YearAsString YEAR

endif

    DC.B 0

endif
    
```

When assembled with the option `-dADD_COPYRIGHT -dYEAR=2005`, the code in the following listing is generated:

### Listing: Generated list file

```

1 1          YearAsString: MACRO
2 2          DC.B $30+(\1 /1000)%10
3 3          DC.B $30+(\1 / 100)%10
4 4          DC.B $30+(\1 / 10)%10
5 5          DC.B $30+(\1 / 1)%10
6 6          ENDM
7 7
8 8          0000 0001  ifdef ADD_COPYRIGHT
9 9          ORG $1000
10 10 a001000 436F 7079  DC.B "Copyright by "
    
```

## List of Assembler Option

```

001004 7269 6768
001008 7420 6279
00100C 20
11 11 a00100D 4A6F 686E DC.B "John Doe"
001011 2044 6F65
12 12 0000 0001 ifdef YEAR
13 13 a001015 2031 3939 DC.B " 1999-"
001019 392D
14 14 YearAsString YEAR
15 2m a00101B 32 + DC.B $30+(YEAR /1000)%10
16 3m a00101C 30 + DC.B $30+(YEAR / 100)%10
17 4m a00101D 30 + DC.B $30+(YEAR / 10)%10
18 5m a00101E 31 + DC.B $30+(YEAR / 1)%10
19 15 endif
20 16 a00101F 00 DC.B 0
21 17 endif

```

## 6.3.9 -DefLabel: Improves support for data allocation directives

### Group

Input

### Scope

Assembly Unit

### Syntax

```
-DefLabel
```

### Arguments

None

### Default

None

### Description

Improves support for data allocation directives. On passing this option, the data directives (not associated to any label) get associated with previous defined labels (if exists). This inhibits the emission of temporary variables ( `VARx` ) by assembler.

## Example

The following listing shows the example for the `-DefLabel`.

### Listing: Example -DefLabel

```
MySection: SECTION
TTab_TIT_45:      ; Modo 5

        DC.B      $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FA,$90,$20,$00,$01,$4F,$FF
        , $FB,$FF,$FF,$FF,$FF,$FC

        DC.B      $C1,$FF,$FF,$FF,$FF,$FF,$FF,$F7,$64,$88,$00,$14,$BF,$FF
        , $F5,$FF,$EF,$B8,$3F,$FE

        DC.B 5

        DC.W 3

mainLoop:

        RTS

With
-DefLabel option
OFF:

Output:

9-VAR00001      0   14 LOCAL FUNC      9 (MySection)
10-VAR00002     14  14 LOCAL FUNC      9 (MySection)
11-VAR00003     28   1 LOCAL  FUNC      9 (MySection)
12-VAR00004     29   3 LOCAL  FUNC      9 (MySection)

With
-DefLabel option
ON:

Output:

9-TTab_TIT_45   0   29 LOCAL FUNC      9 (MySection)
10-VAR00001     29   3 LOCAL  FUNC      9 (MySection)
```

The input file when assembled with option `-DefLabel`, allocates the `DC.B` directives to symbol `TTab_TIT_45` and `DC.W` to dummy variable `VAR00001`.

## 6.3.10 -Env: Set environment variable

### Group

## List of Assembler Option

Host

### Scope

Assembly Unit

### Syntax

```
-Env<EnvironmentVariable>=<VariableSetting>
```

### Arguments

<EnvironmentVariable>: Environment variable to be set

<VariableSetting>: Setting of the environment variable

### Default

None

### Description

This option sets an environment variable.

### Example

```
ASMOPTIONS=-EnvOBJPATH=\sources\obj
```

This is the same as:

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

in the `default.env` file.

### See also

[Environment variables details](#)

## 6.3.11 -F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format

### Group

Output

## Scope

Application

## Syntax

```
-F(h|2o|A2o|2|A2)
```

## Arguments

h: HIWARE object-file format; this is the default

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

A2o: Compatible ELF/DWARF 2.0 absolute-file format

2: ELF/DWARF 2.0 object-file format

A2: ELF/DWARF 2.0 absolute-file format

## Default

```
-F2
```

## Description

Defines the format for the output file generated by the Assembler:

- With the `-Fh` option set, the Assembler uses a proprietary (HIWARE) object-file format.
- With the `-F2` option set, the Assembler produces an ELF/DWARF object file. This object-file format may also be supported by other Compiler or Assembler vendors.
- With the `-FA2` option set, the Assembler produces an ELF/DWARF absolute file. This file format may also be supported by other Compiler or Assembler vendors.

Note that the ELF/DWARF 2.0 file format has been updated in the current version of the Assembler. If you are using HI-WAVE version 5.2 (or an earlier version), `-F2o` or `-FA2o` must be used to generate the ELF/DWARF 2.0 object files which can be loaded in the debugger.

## Example

```
ASMOPTIONS=-F2
```

## NOTE

For the RS08 the HIWARE object file format is not available.

## 6.3.12 -H: Short Help

### Group

Various

### Scope

None

### Syntax

-H

### Arguments

None

### Default

None

### Description

The `-H` option causes the Assembler to display a short list (i.e., help list) of available options within the assembler window. Options are grouped into Output, Input, Language, Host, Code Generation, Messages, and Various.

No other option or source files should be specified when the `-H` option is invoked.

### Example

The following listing is a portion of the list produced by the `-H` option:

#### Listing: Example Help listing

```
...  
MESSAGE:  
  
-N          Show notification box in case of errors  
  
-NoBeep    No beep in case of an error  
  
-W1        Do not print INFORMATION messages  
  
-W2        Do not print INFORMATION or WARNING messages  
  
-WErrFile  Create "err.log" Error File  
  
...
```



### 6.3.13 -I: Include file path

#### Group

Input

#### Scope

None

#### Syntax

```
-I<path>
```

#### Arguments

<path>: File path to be used for includes

#### Default

None

#### Description

With the `-I` option it is possible to specify a file path used for include files.

#### Example

```
-Id:\mySources\include
```

### 6.3.14 -L: Generate a listing file

#### Group

Output

#### Scope

Assembly unit

#### Syntax

`-L [=<dest>]`

## Arguments

`<dest>`: the name of the listing file to be generated.

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

## Default

No generated listing file

## Description

Switches on the generation of the listing file. If `dest` is not specified, the listing file will have the same name as the source file, but with extension `*.lst`. The listing file contains macro definition, invocation, and expansion lines as well as expanded include files.

## Example

```
ASMOPTIONS=-L
```

In the following example of assembly code, the `cpChar` macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When the `-L` option is specified, the portion of assembly source code in the following listing, together with the code from an include file ( [Listing: Example source code from an include file](#)) generates the output listing in [Listing: Assembly output listing](#).

### Listing: Example assembly source code

```

        XDEF Start
MyData: SECTION
char1:  DS.B  1
char2:  DS.B  1
        INCLUDE "macro.inc"
CodeSec: SECTION
Start:
        cpChar char1, char2
        NOP

```

### Listing: Example source code from an include file

```
cpChar: MACRO
```

```

LDA \1
STA \2
ENDM
    
```

### Listing: Assembly output listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			XDEF Start
2	2			MyData: SECTION
3	3	000000		char1: DS.B 1
4	4	000001		char2: DS.B 1
5	5			INCLUDE "macro.inc"
6	1i			cpChar: MACRO
7	2i			LDA \1
8	3i			STA \2
9	4i			ENDM
10	6			CodeSec: SECTION
11	7			Start:
12	8			cpChar char1, char2
13	2m	000000	C6 xxxx +	LDA char1
14	3m	000003	C7 xxxx +	STA char2
15	9	000006	9D	NOP

The Assembler stores the content of included files in the listing file. The Assembler also stores macro definitions, invocations, and expansions in the listing file.

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

Assembler options:

- [-Lasmc: Configure listing file](#)
- [-Lasms: Configure the address size in the listing file](#)
- [-Lc: No Macro call in listing file](#)
- [-Ld: No macro definition in listing file](#)
- [-Le: No Macro expansion in listing file](#)
- [-Li: No included file in listing file](#)

## 6.3.15 -Lasmc: Configure listing file

### Group

Output

### Scope

Assembly unit

### Syntax

```
-Lasmc={s|r|m|l|k|i|c|a}
```

### Arguments

s - Do not write the source column

r - Do not write the relative column (Rel.)

m - Do not write the macro mark

l - Do not write the address (Loc)

k - Do not write the location type

i - Do not write the include mark column

c - Do not write the object code

a - Do not write the absolute column (Abs.)

### Default

Write all columns.

### Description

The default-configured listing file shows a lot of information. With this option, the output can be reduced to columns which are of interest. This option configures which columns are printed in a listing file. To configure which lines to print, see the following assembler options: [-Lc: No Macro call in listing file](#), [-Ld: No macro definition in listing file](#), [-Le: No Macro expansion in listing file](#), and [-Li: No included file in listing file](#).

### Example

For the following assembly source code, the Assembler generates the default-configured output listing, as shown in the following listing:

```
DC.B "Hello World"
```

```
DC.B 0
```

### Listing: Example assembler output listing

Abs.	Rel.	Loc	Obj. code	Source line
1	1	000000	4865 6C6C	DC.B "Hello World"
		000004	6F20 576F	
		000008	726C 64	
2	2	00000B	00	DC.B 0

In order to get this output without the source file line numbers and other irrelevant parts for this simple `DC.B` example, the following option is added:

`-Lasmc=ramki`. This generates the output listing as shown in the following listing:

### Listing: Example output listing

Loc	Obj. code	Source line
000000	4865 6C6C	DC.B "Hello World"
000004	6F20 576F	
000008	726C 64	
00000B	00	DC.B 0

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

Assembler options:

- [-L: Generate a listing file](#)
- [-Lc: No Macro call in listing file](#)
- [-Ld: No macro definition in listing file](#)
- [-Le: No Macro expansion in listing file](#)
- [-Li: No included file in listing file](#)
- [-Lasms: Configure the address size in the listing file](#)

## 6.3.16 -Lasms: Configure the address size in the listing file

### Group

Output

### Scope

Assembly unit

### Syntax

```
-Lasms{1|2|3|4}
```

### Arguments

- 1 - The address size is xx
- 2 - The address size is xxxx
- 3 - The address size is xxxxxx
- 4 - The address size is xxxxxxxx

### Default

```
-Lasms3
```

### Description

The default-configured listing file shows a lot of information. With this option, the size of the address column can be reduced to the size of interest. To configure which columns are printed, see the [-Lasmc: Configure listing file](#) option. To configure which lines to print, see the [-Lc: No Macro call in listing file](#), [-Ld: No macro definition in listing file](#), [-Le: No Macro expansion in listing file](#), and [-Li: No included file in listing file](#) assembler options.

### Example

For the following instruction:

NOP

the Assembler generates this default-configured output listing as listed below:

### Listing: Example assembler output listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1	000000	XX	NOP

In order to change the size of the address column the following option is added:

`-Lasms1`. This changes the address size to two digits.

### Listing: Example assembler output listing configured with `-Lasms1`

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1	00	XX	NOP

### See also

[Assembler Listing File](#) chapter

### Assembler options :

- [-Lasmc](#): Configure listing file
- [-L](#): Generate a listing file
- [-Lc](#): No Macro call in listing file
- [-Ld](#): No macro definition in listing file
- [-Le](#): No Macro expansion in listing file
- [-Li](#): No included file in listing file

## 6.3.17 -Lc: No Macro call in listing file

### Group

Output

### Scope

Assembly unit

### Syntax

-Lc

## Arguments

none

## Default

none

## Description

Switches on the generation of the listing file, but macro invocations are not present in the listing file. The listing file contains macro definition and expansion lines as well as expanded include files.

## Example

```
ASMOPTIONS=-Lc
```

In the following example of assembly code, the `cpChar` macro accept two parameters. The macro copies the value of the first parameter to the second one.

When the `-Lc` option is specified, the following portion of assembly source code in the following listing, along with additional source code (*Listing: Example source code from the macro.inc file*) from the `macro.inc` include file generates the output in the assembly listing file (*Listing: Output assembly listing*).

### Listing: Example assembly source code

```

MyData: XDEF Start
        SECTION

char1:  DS.B  1

char2:  DS.B  1

        INCLUDE "macro.inc"

CodeSec: SECTION

Start:

        cpChar char1, char2

        NOP

```

### Listing: Example source code from the macro.inc file

```

cpChar:  MACRO
        LDA \1

```



```

        STA \2
    ENDM
    
```

### Listing: Output assembly listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			XDEF Start
2	2			MyData: SECTION
3	3	000000		char1: DS.B 1
4	4	000001		char2: DS.B 1
5	5			INCLUDE "macro.inc"
6	1i			cpChar: MACRO
7	2i			LDA \1
8	3i			STA \2
9	4i			ENDM
10	6			CodeSec: SECTION
11	7			Start:
13	2m	000000	C6 xxxxx +	LDA char1
14	3m	000003	C7 xxxxx +	STA char2
15	9	000006	9D	NOP

The Assembler stores the content of included files in the listing file. The Assembler also stores macro definitions, invocations, and expansions in the listing file.

The listing file does not contain the line of source code that invoked the macro.

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

Assembler options:

- [-L: Generate a listing file](#)
- [-Ld: No macro definition in listing file](#)
- [-Le: No Macro expansion in listing file](#)
- [-Li: No included file in listing file](#)

### 6.3.18 -Ld: No macro definition in listing file

## Group

Output

## Scope

Assembly unit

## Syntax

-Ld

## Arguments

None

## Default

None

## Description

Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.

## Example

```
ASMOPTIONS=-Ld
```

In the following example of assembly code, the `cpChar` macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When the `-Ld` option is specified, the assembly source code in the following listing along with additional source code (*Listing: Example source code from an include file*) from the `macro.inc` file generates an assembler output listing (*Listing: Example assembler output listing*) file:

### Listing: Example assembly source code

```
MyData: XDEF Start
        SECTION

char1:  DS.B 1

char2:  DS.B 1

        INCLUDE "macro.inc"

CodeSec: SECTION
```

```
Start:
    cpChar char1, char2
    NOP
```

### Listing: Example source code from an include file

```
cpChar: MACRO
        LDA  \1
        STA  \2
    ENDM
```

### Listing: Example assembler output listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			XDEF Start
2	2			MyData: SECTION
3	3	000000		char1: DS.B 1
4	4	000001		char2: DS.B 1
5	5			INCLUDE "macro.inc"
6	1i			cpChar: MACRO
10	6			CodeSec: SECTION
11	7			Start:
12	8			cpChar char1, char2
13	2m	000000	C6 xxxx +	LDA char1
14	3m	000003	C7 xxxx +	STA char2
15	9	000006	9D	NOP

The Assembler stores that content of included files in the listing file. The Assembler also stores macro invocation and expansion in the listing file.

The listing file does not contain the source code from the macro definition.

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

Assembler options:

- [-L: Generate a listing file](#)
- [-Lc: No Macro call in listing file](#)
- [-Le: No Macro expansion in listing file](#)
- [-Li: No included file in listing file](#)

## 6.3.19 -Le: No Macro expansion in listing file

### Group

Output

### Scope

Assembly unit

### Syntax

```
-Le
```

### Arguments

None

### Default

None

### Description

Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.

### Example

```
ASMOPTIONS=-Le
```

In the following example of assembly code, the `cpChar` macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When the `-Le` option is specified, the assembly code in the following listing along with additional source code (*Listing: Example source code from an included file*) from the `macro.inc` file generates an assembly output listing file (*Listing: Example assembler output listing*):

### Listing: Example assembly source code

```
                XDEF Start  
MyData:        SECTION
```

```

char1:  DS.B  1
char2:  DS.B  1
        INCLUDE "macro.inc"
CodeSec: SECTION
Start:
        cpChar char1, char2
        NOP
    
```

### Listing: Example source code from an included file

```

cpChar:  MACRO
        LDA  \1
        STA  \2
        ENDM
    
```

### Listing: Example assembler output listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			XDEF Start
2	2			MyData: SECTION
3	3	000000		char1: DS.B 1
4	4	000001		char2: DS.B 1
5	5			INCLUDE "macro.inc"
6	1i			cpChar: MACRO
7	2i			LDA \1
8	3i			STA \2
9	4i			ENDM
10	6			CodeSec: SECTION
11	7			Start:
12	8			cpChar char1, char2
15	9	000006 9D		NOP

The Assembler stores the content of included files in the listing file. The Assembler also stores the macro definition and invocation in the listing file.

The Assembler does not store the macro expansion lines in the listing file.

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

- L: Generate a listing file
- Lc: No Macro call in listing file
- Ld: No macro definition in listing file
- Li: No included file in listing file

### 6.3.20 -Li: No included file in listing file

#### Group

Output

#### Scope

Assembly unit

#### Syntax

```
-Li
```

#### Arguments

None

#### Default

None

#### Description

Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

#### Example

```
ASMOPTIONS=-Li
```

In the following example of assembly code, the `cpChar` macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When `-Li` option is specified, the assembly source code in the following listing along with additional source code (*Listing: Example source code in an include file*) from the `macro.inc` file generates the following output in the assembly listing file:

## Listing: Example assembly source code

```

XDEF Start
MyData: SECTION

char1: DS.B 1

char2: DS.B 1

INCLUDE "macro.inc"

CodeSec: SECTION

Start:

    cpChar char1, char2

    NOP
    
```

## Listing: Example source code in an include file

```

cpChar: MACRO
    LDA  \1

    STA  \2

ENDM
    
```

## Listing: Example assembler output listing

Abs.	Rel.	Loc	Obj.	code	Source line
----	----	-----	-----	-----	-----
1	1				XDEF Start
2	2				MyData: SECTION
3	3	000000			char1: DS.B 1
4	4	000001			char2: DS.B 1
5	5				INCLUDE "macro.inc"
10	6				CodeSec: SECTION
11	7				Start:
12	8				cpChar char1, char2
13	2m	000000	C6	xxxx +	LDA char1
14	3m	000003	C7	xxxx +	STA char2
15	9	000006	9D		NOP

The Assembler stores the macro definition, invocation, and expansion in the listing file.

The Assembler does not store the content of included files in the listing file.

For a detailed description of the listing file, see the [Assembler Listing File](#) chapter.

### See also

Assembler options:

- [-L](#): Generate a listing file
- [-Lc](#): No Macro call in listing file
- [-Ld](#): No macro definition in listing file
- [-Le](#): No Macro expansion in listing file

## 6.3.21 -Lic: License information

### Group

Various

### Scope

None

### Syntax

```
-Lic
```

### Arguments

None

### Default

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

```
ASMOPTIONS=-Lic
```

### See also

Assembler options:

- [-LicA](#): License information about every feature in directory



- [-LicBorrow](#): Borrow license feature
- [-LicWait](#): Wait until floating license is available from floating License Server

### 6.3.22 **-LicA: License information about every feature in directory**

#### **Group**

Various

#### **Scope**

None

#### **Syntax**

```
-LicA
```

#### **Arguments**

None

#### **Default**

None

#### **Description**

The `-LicA` option prints the license information of every tool or DLL in the directory where the executable is (e.g., if tool or feature is a demo version or a full version). Because the option has to analyze every single file in the directory, this may take a long time.

#### **Example**

```
ASMOPTIONS=-LicA
```

#### **See also**

#### **Assembler options :**

- [-Lic: License information](#)

- [-LicBorrow: Borrow license feature](#)
- [-LicWait: Wait until floating license is available from floating License Server](#)

### 6.3.23 -LicBorrow: Borrow license feature

#### Group

Host

#### Scope

None

#### Syntax

```
-LicBorrow<feature> [;<version>] :<Date>
```

#### Arguments

<feature>: the feature name to be borrowed (e.g., HI100100).

<version>: optional version of the feature to be borrowed (e.g., 3.000).

<date>: date with optional time until when the feature shall be borrowed (e.g., 15-Mar-2005:18:35).

#### Default

None

#### Defines

None

#### Pragmas

None

#### Description

This option lets you borrow a license feature until a given date/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 is aware of the version). However, if you want to borrow any feature, you need to specify the feature's version number.

You can check the status of currently borrowed features in the tool's *About* box.

### NOTE

You only can borrow features if you have a floating license and if your floating license is enabled for borrowing. See the provided FLEXlm documentation about details on borrowing.

### Example

```
-LicBorrowHI100100;3.000:12-Mar-2005:18:25
```

### See also

Assembler options:

- [-Lic: License information](#)
- [-LicA: License information about every feature in directory](#)
- [-LicWait: Wait until floating license is available from floating License Server](#)

## 6.3.24 -LicWait: Wait until floating license is available from floating License Server

### Group

Host

### Scope

None

### Syntax

```
-LicWait
```

### Arguments

None

## Default

None

## Description

If a license is not available from the floating license server, then the default condition is that the application will immediately return. With the `-LicWait` assembler option set, the application will wait (blocking) until a license is available from the floating license server.

## Example

```
ASMOPTIONS=-LicWait
```

## See also

Assembler options:

- [-Lic: License information](#)
- [-LicA: License information about every feature in directory](#)
- [-LicBorrow: Borrow license feature](#)

## 6.3.25 -LI: Show label statistics

### Group

Output

### Syntax

```
-LI
```

### Arguments

None

### Description

It displays label statistics in the list file. The option gives the gain in terms of code size for a label if moved to SHORT or TINY section.

### Example:

```

Test.asm:

TINY_RAM_VARS: SECTION  RS08_SHORT;Insert your data
definition here

tmp:          DS.B  1..

FiboLoop:     STA    tmp ; store last

RTS

Test.lst:

Freescale Assembler

Ind.  Name          tiny          short
----  ----          ----          -----
      1 tmp                1            1

```

### 6.3.26 -M (-Ms, -Mt): Memory model

#### Group

Code Generation

#### Scope

Application

#### Syntax

```
-M(s|b|t)
```

#### Arguments

s: small memory model

t: tiny memory model

#### Default

```
-Ms
```

#### Description

The Assembler for the MC68HC(S)08 supports two different memory models. The default is the small memory model, which corresponds to the normal setup, i.e., a 64kB code-address space. The tiny memory model corresponds to the situation where the default RAM is in the zero page.

### NOTE

For the Assembler, the memory model does not matter at all. The memory model is used by the compiler to specify the default allocation of variable and functions. The Assembler has this option only to generate "compatible" object files for the memory model consistency check of the linker.

### NOTE

In the tiny memory model, the default for the compiler is to use zero-page addressing. The default for the Assembler is to still use extended-addressing modes. See the [Using the direct addressing mode to access symbols](#) section to see how to generate zero-page accesses.

## Example

```
ASMOPTIONS=-Mt
```

## 6.3.27 -MacroNest: Configure maximum macro nesting

### Group

Language

### Scope

Assembly Unit

### Syntax

```
-MacroNest<Value>
```

### Arguments

<Value>: max. allowed nesting level

### Default

3000

## Description

This option controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

## Example

See the description of message A1004 for an example.

### 6.3.28 Message A1004 (available in the Online Help)

### 6.3.29 -MCUasm: Switch compatibility with MCUasm ON

#### Group

Various

#### Scope

Assembly Unit

#### Syntax

```
-MCUasm
```

#### Arguments

None

#### Default

None

#### Description

This switches `ON` compatibility mode with the MCUasm Assembler. Additional features supported, when this option is activated are enumerated in the [MCUasm Compatibility](#) chapter in the Appendices.

#### Example

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014

ASMOPTIONS=-MCUasm

### 6.3.30 -MMU: Enable Memory Management Unit (MMU) Support

#### Group

CODE GENERATION

#### Scope

Assembly Unit

#### Syntax

-MMU

#### Arguments

None

#### Default

None

#### Defines

`__MMU__`

#### Pragmas

None

#### Description

This option enables code banking and CALL and RTC instructions are available. It can be used only when `-CS08` is enabled.

### 6.3.31 -N: Display notify box

#### Group

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014



Messages

### Scope

Assembly Unit

### Syntax

-N

### Arguments

None

### Default

None

### Description

Makes the Assembler display an alert box if there was an error during assembling. This is useful when running a makefile (please see the manual about *Build Tools*) because the Assembler waits for the user to acknowledge the message, thus suspending makefile processing. (The 'N' stands for "Notify".)

This feature is useful for halting and aborting a build using the Make Utility.

### Example

```
ASMOPTIONS=-N
```

If an error occurs during assembling, an alert dialog box will be opened.

## 6.3.32 -NoBeep: No beep in case of an error

### Group

Messages

### Scope

Assembly Unit

### Syntax

-NoBeep

### Arguments

None

### Default

None

### Description

Normally there is a `beep' notification at the end of processing if there was an error. To have a silent error behavior, this `beep' may be switched off using this option.

### Example

```
ASMOPTIONS=-NoBeep
```

## 6.3.33 -NoDebugInfo: No debug information for ELF/DWARF files

### Group

Language

### Scope

Assembly Unit

### Syntax

-NoDebugInfo

### Arguments

None

### Default

None

### Description

By default, the Assembler produces debugging info for the produced ELF/DWARF files. This can be switched off with this option.

## Example

```
ASMOPTIONS=-NoDebugInfo
```

## 6.3.34 -NoEnv: Do not use environment

### Group

Startup (This option cannot be specified interactively.)

### Scope

Assembly Unit

### Syntax

```
-NoEnv
```

### Arguments

None

### Default

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 does not use any environment ( `default.env`, `project.ini` or tips file).

### Example

```
xx  
.exe -NoEnv
```

(Use the actual executable name instead of *xx*)

## See also

[Environment](#) chapter

## 6.3.35 -ObjN: Object filename specification

### Group

Output

### Scope

Assembly Unit

### Syntax

```
-ObjN<FileName>
```

### Arguments

<FileName>: Name of the binary output file generated.

### Default

-ObjN%n.o when generating a relocatable file or

-ObjN%n.abs when generating an absolute file.

### Description

Normally, the object file has the same name than the processed source file, but with the .o extension when relocatable code is generated or the .abs extension when absolute code is generated. This option allows a flexible way to define the output filename. The modifier %n can also be used. It is replaced with the source filename. If <file> in this option contains a path (absolute or relative), the OBJPATH environment variable is ignored.

### Example

For ASMOPTIONS=-ObjNa.out, the resulting object file will be a.out. If the OBJPATH environment variable is set to \src\obj, the object file will be \src\obj\a.out.

```
For  
fibo.c -ObjN%n.obj, the resulting object file will be
```

```
fibonacci.obj.
```

For `myfile.c -ObjN..\objects\_%n.obj`, the object file will be named relative to the current directory to `...\objects\_myfile.obj`. Note that the environment variable `OBJPATH` is ignored, because `<file>` contains a path.

### See also

[OBJPATH: Object file path](#) environment variable

## 6.3.36 -Prod: Specify project file at startup

### Group

None (This option cannot be specified interactively.)

### Scope

None

### Syntax

```
-Prod=<file>
```

### Arguments

`<file>`: name of a project or project directory

### Default

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 configuration file. When the filename does only contain a directory, the default name `project.ini` is appended. When the loading fails, a message box appears.

### Example

```
assembler.exe -Prod=project.ini
```

(Use the Assembler's executable name instead of `assembler`.)

### See also

[Environment](#) chapter

## 6.3.37 -Struct: Support for structured types

### Group

Input

### Scope

Assembly Unit

### Syntax

```
-Struct
```

### Arguments

None

### Default

None

### Description

When this option is activated, the Macro Assembler also support the definition and usage of structured types. This is interesting for application containing both ANSI-C and Assembly modules.

### Example

```
ASMOPTIONS=-Struct
```

### See also

[Mixed C and Assembler Applications](#) chapter

## 6.3.38 -V: Prints the Assembler version

### Group

Various

### Scope

None

### Syntax

-v

### Arguments

None

### Default

None

### Description

Prints the Assembler version and the current directory.

#### NOTE

Use this option to determine the current directory of the Assembler.

### Example

-V produces the following listing:

#### Listing: Example of a version listing

```
Command Line '-v'
Assembler V-5.0.8, Jul  7 2005

Directory: C:\Freescale\demo

Common Module V-5.0.7, Date Jul  7 2005

User Interface Module, V-5.0.17, Date Jul  7 2005

Assembler Kernel, V-5.0.13, Date Jul  7 2005

Assembler Target, V-5.0.8, Date Jul  7 2005
```

## 6.3.39 -View: Application standard occurrence

### Group

Host

### Scope

Assembly Unit

### Syntax

```
-View<kind>
```

### Arguments

<kind> is one of the following:

- "Window": Application window has the default window size.
- "Min": Application window is minimized.
- "Max": Application window is maximized.
- "Hidden": Application window is not visible (only if there are arguments).

### Default

Application is started with arguments: `Minimized`.

Application is started without arguments: `Window`.

### Description

Normally, the application is started with a normal window if no arguments are given. If the application is started with arguments (e.g., from the Maker to assemble, compile, or link a file), then the application is running minimized to allow for batch processing. However, the application's window behavior may be specified with the View 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, for example, if you are using the [-N: Display notify box](#) assembler option, a dialog box is still possible.

### Example



```
C:\Freescale\prog\linker.exe -ViewHidden fibo.prm
```

## 6.3.40 -W1: No information messages

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-W1
```

### Arguments

None

### Default

None

### Description

Inhibits the Assembler's printing INFORMATION messages. Only WARNING and ERROR messages are written to the error listing file and to the assembler window.

### Example

```
ASMOPTIONS=-W1
```

## 6.3.41 -W2: No information and warning messages

### Group

Messages

### Scope

Assembly Unit

## Syntax

-W2

## Arguments

None

## Default

None

## Description

Suppresses all messages of INFORMATION or WARNING types. Only ERROR messages are written to the error listing file and to the assembler window.

## Example

```
ASMOPTIONS=-W2
```

## 6.3.42 -WErrFile: Create "err.log" error file

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WErrFile(On|Off)
```

### Arguments

None

### Default

An `err.log` file is created or deleted.

## Description

The error feedback from the Assembler to called tools is now done with a return code. In 16-bit Windows environments this was not possible. So in case of an error, an "err.log" file with the numbers of written errors was used to signal any errors. To indicate no errors, the "err.log" file would be deleted. Using UNIX or WIN32, a return code is now available. Therefore, this file is no longer needed when only UNIX or WIN32 applications are involved. To use a 16-bit Maker with this tool, an error file must be created in order to signal any error.

## Example

- -WErrFileOn

err.log is created or deleted when the application is finished.

- -WErrFileOff

existing err.log is not modified.

## See also

[-WStdout: Write to standard output](#)

[-WOutFile: Create error listing file](#)

## 6.3.43 -Wmsg8x3: Cut filenames in Microsoft format to 8.3

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-Wmsg8x3
```

### Default

None

### Description

## List of Assembler Option

Some editors (e.g., early versions of WinEdit) are expecting the filename in the Microsoft message format in a strict 8.3 format. That means the filename can have at most 8 characters with not more than a 3-character extension. Using a newer Windows OS, longer file names are possible. With this option the filename in the Microsoft message is truncated to the 8.3 format.

### Example

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

With the `-Wmsg8x3` option set, the above message will be

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

### See also

- [-WmsgFb \(-WmsgFbv, -WmsgFbm\): Set message file format for batch mode](#)
- [-WmsgFi \(-WmsgFiv, -WmsgFim\): Set message file format for interactive mode](#)
- [-WmsgFoi: Message format for interactive mode](#)
- [-WmsgFob: Message format for batch mode Option](#)
- [-WmsgFonp: Message format for no position information](#)

## 6.3.44 -WmsgCE: RGB color for error messages

### Group

Messages

### Scope

Compilation Unit

### Syntax

```
-WmsgCE<  
RGB>
```

### Arguments

<RGB>: 24-bit RGB (red green blue) value.

## Default

```
-WmsgCE16711680 (  
rFF  
g00  
b00, red)
```

## Description

It is possible to change the error message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

## Example

`-WmsgCE255` changes the error messages to blue.

## 6.3.45 -WmsgCF: RGB color for fatal messages

### Group

Messages

### Scope

Compilation Unit

### Syntax

```
-WmsgCF<  
RGB>
```

### Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

### Default

```
-WmsgCF8388608 (  
r80  
g00  
b00, dark red)
```

## Description

It is possible to change the fatal message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

### Example

`-WmsgCF255` changes the fatal messages to blue.

## 6.3.46 `-WmsgCI`: RGB color for information messages

### Group

Messages

### Scope

Compilation Unit

### Syntax

```
-WmsgCI<
RGB>
```

### Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

### Default

```
-WmsgCI32768 (
r00
g80
b00, green)
```

### Description

It is possible to change the information message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

### Example

`-WmsgCI255` changes the information messages to blue.

## 6.3.47 -WmsgCU: RGB color for user messages

### Group

Messages

### Scope

Compilation Unit

### Syntax

```
-WmsgCU<  
RGB>
```

### Arguments

<RGB>: 24-bit RGB (red green blue) value.

### Default

```
-WmsgCU0 (  
r00  
g00  
b00, black)
```

### Description

It is possible to change the user message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

### Example

`-WmsgCU255` changes the user messages to blue.

## 6.3.48 -WmsgCW: RGB color for warning messages

### Group

Messages

### Scope

Compilation Unit

## Syntax

-WmsgCW<

RGB>

## Arguments

<RGB>: 24-bit RGB (red green blue) value.

## Default

```
-WmsgCW255 (
r00
g00
bFF, blue)
```

## Description

It is possible to change the warning message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

## Example

-WmsgCW0 changes the warning messages to black.

## 6.3.49 -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode

### Group

Messages

### Scope

Assembly Unit

### Syntax

-WmsgFb [v|m]

### Arguments

v: Verbose format.



m: Microsoft format.

## Default

-WmsgFbm

## Description

The Assembler can be started with additional arguments (e.g., files to be assembled together with assembler options). If the Assembler has been started with arguments (e.g., from the *Make tool*), the Assembler works in the batch mode. That is, no assembler window is visible and the Assembler terminates after job completion.

If the Assembler is in batch mode, the Assembler messages are written to a file and are not visible on the screen. This file only contains assembler messages (see examples below).

The Assembler uses a *Microsoft* message format as the default to write the assembler messages (errors, warnings, or information messages) if the Assembler is in the batch mode.

With this option, the default format may be changed from the *Microsoft* format (with only line information) to a more verbose error format with line, column, and source information.

## Example

Assume that the assembly source code in the following listing is to be assembled in the batch mode.

### Listing: Example assembly source code

```
var1:   equ 5
var2:   equ 5

    if (var1=var2)
        NOP
    endif
endif
```

The Assembler generates the error output, as shown in the following listing, in the assembler window if it is running in batch mode:

### Listing: Example error listing in the Microsoft (default) format for batch mode

```
X:\TW2.ASM(12):ERROR: Conditional else not allowed here.
```

## List of Assembler Option

If the format is set to verbose, more information is stored in the file:

### Listing: Example error listing in the verbose format for batch mode

```
ASMOPTIONS=-WmsgFbv
>> in "C:\tw2.asm", line 6, col 0, pos 81

    endif
^

ERROR A1001: Conditional else not allowed here
```

### See also

[ERRORFILE: Filename specification error](#)

[-WmsgFi \(-WmsgFiv, -WmsgFim\): Set message file format for interactive mode](#)

[-WmsgFob: Message format for batch mode](#)

[-WmsgFoi: Message format for interactive mode](#)

[-WmsgFonf: Message format for no file information](#)

[-WmsgFonp: Message format for no position information](#)

## 6.3.50 -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WmsgFi [v|m]
```

### Arguments

v: Verbose format.

m: Microsoft format.

### Default

-WmsgFiv

## Description

If the Assembler is started without additional arguments (e.g., files to be assembled together with Assembler options), the Assembler is in the interactive mode (that is, a window is visible).

While in interactive mode, the Assembler uses the default verbose error file format to write the assembler messages (errors, warnings, information messages).

Using this option, the default format may be changed from verbose (with source, line and column information) to the *Microsoft* format (which displays only line information).

## NOTE

Using the Microsoft format may speed up the assembly process because the Assembler has to write less information to the screen.

## Example

If the Assembler is running in interactive mode, the default error output is shown in the assembler window as in the following listing.

### Listing: Example error listing in the default mode for interactive mode

```
>> in "X:\TWE.ASM", line 12, col 0, pos 215
      endif

      endif
^
ERROR A1001: Conditional else not allowed here
```

Setting the format to Microsoft, less information is displayed:

### Listing: Example error listing in Microsoft format for interactive mode

```
ASMOPTIONS=-WmsgFim
X:\TWE.ASM(12): ERROR: conditional else not allowed here
```

## See also

[ERRORFILE: Filename specification error](#) environment variable

Assembler options:

- [-WmsgFb \(-WmsgFbv, -WmsgFbm\)](#): Set message file format for batch mode
- [-WmsgFob](#): Message format for batch mode

- -WmsgFoi: Message format for interactive mode
- -WmsgFonf: Message format for no file information
- -WmsgFonp: Message format for no position information

### 6.3.51 -WmsgFob: Message format for batch mode

#### Group

Messages

#### Scope

Assembly Unit

#### Syntax

-WmsgFob<string>

#### Arguments

<string>: format string (see below).

#### Default

-WmsgFob"%f%e(%l): %K %d: %m\n"

#### Description

With this option it is possible to modify the default message format in the batch mode. The formats in in the following listing are supported (assumed that the source file is x:\Freescale\sourcefile.asm).

#### Listing: Supported formats for messages in the batch mode

Format	Description	Example
%s	Source Extract	
%p	Path	x:\Freescale\
%f	Path and name	x:\Freescale\sourcefile
%n	Filename	sourcefile
%e	Extension	.asmx
%N	File (8 chars)	sourcefi

%E	Extension (3 chars)	.asm
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	A1051
%m	Message	text
%%	Percent	%
\n	New line	

## Example

```
ASMOPTIONS=-WmsgFob"%f%e(%l): %k %d: %m\n"
```

produces a message, displayed in in the following listing, using the format in in the above listing. The options are set for producing the path of a file with its filename, extension, and line.

### Listing: Error message

```
x:\Freescale\sourcefile.asm(3): error A1051: Right parenthesis
expected
```

### See also

Assembler options:

- [-WmsgFb \(-WmsgFbv, -WmsgFbm\)](#): Set message file format for batch mode
- [-WmsgFi \(-WmsgFiv, -WmsgFim\)](#): Set message file format for interactive mode
- [-WmsgFoi](#): Message format for interactive mode
- [-WmsgFonf](#): Message format for no file information
- [-WmsgFonp](#): Message format for no position information

## 6.3.52 -WmsgFoi: Message format for interactive mode

### Group

Messages

### Scope

## Assembly Unit

### 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"
```

### Description

With this option it is possible modify the default message format in interactive mode. The following formats are supported (supposed that the source file is x:\Freescale\sourcefile.asm):

#### Listing: Supported message formats - interactive mode

Format	Description	Example
%s	Source Extract	
%p	Path	x:\Freescale\
%f	Path and name	x:\Freescale\sourcefile
%n	Filename	sourcefile
%e	Extension	.asmx
%N	File (8 chars)	sourcefi
%E	Extension (3 chars)	.asm
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	A1051
%m	Message	text
%%	Percent	%

\n      New line

## Example

```
ASMOPTIONS=-WmsgFoi"%f%e(%l): %k %d: %m\n"
```

produces a message in following listed format:

### Listing: Error message resulting from the statement above

```
x:\Freescale\sourcefile.asm(3): error A1051: Right parenthesis  
expected
```

## See also

[ERRORFILE: Filename specification error](#) environment variable

Assembler options:

- [-WmsgFb \(-WmsgFbv, -WmsgFbm\)](#): Set message file format for batch mode
- [-WmsgFi \(-WmsgFiv, -WmsgFim\)](#): Set message file format for interactive mode
- [-WmsgFob](#): Message format for batch mode
- [-WmsgFonf](#): Message format for no file information
- [-WmsgFonp](#): Message format for no position information

## 6.3.53 -WmsgFonf: Message format for no file information

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WmsgFonf<string>
```

### Arguments

<string>: format string (see below)

### Default

## List of Assembler Option

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

### Description

Sometimes there is no file information available for a message (e.g., if a message not related to a specific file). Then this message format string is used. The following formats are supported:

Format	Description	Example
-----		
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L10324
%m	Message	text
%%	Percent	%
\n	New line	

### Example

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

produces a message in following format:

```
information L10324: Linking successful
```



**See also**

[ERRORFILE](#): Filename specification error environment variable

Assembler options:

- [-WmsgFb](#) ([-WmsgFbv](#), [-WmsgFbm](#)): Set message file format for batch mode
- [-WmsgFi](#) ([-WmsgFiv](#), [-WmsgFim](#)): Set message file format for interactive mode
- [-WmsgFob](#): Message format for batch mode
- [-WmsgFoi](#): Message format for interactive mode
- [-WmsgFonp](#): Message format for no position information

**6.3.54 -WmsgFonp: Message format for no position information****Group**

Messages

**Scope**

Assembly Unit

**Syntax**

```
-WmsgFonp<string>
```

**Arguments**

<string>: format string (see below)

**Default**

```
-WmsgFonp"%f%e: %K %d: %m\n"
```

**Description**

Sometimes there is no position information available for a message (e.g., if a message not related to a certain position). Then this message format string is used. The following formats are supported (supposed that the source file is `x:\Freescale\sourcefile.asm`)

**Listing: Supported message formats for when there is no position information**

Format	Description	Example
-----		

## List of Assembler Option

%p	Path	x:\Freescale\
%f	Path and name	x:\Freescale\sourcefile
%n	Filename	sourcefile
%e	Extension	.asmx
%N	File (8 chars)	sourcefi
%E	Extension (3 chars)	.asm
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L10324
%m	Message	text
%%	Percent	%
\n	New line	

## Example

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

produces a message in following format:

```
information L10324: Linking successful
```

## See also

[ERRORFILE: Filename specification error environment variable](#)

Assembler options:

- [-WmsgFb \(-WmsgFbv, -WmsgFbm\): Set message file format for batch mode](#)
- [-WmsgFi \(-WmsgFiv, -WmsgFim\): Set message file format for interactive mode](#)
- [-WmsgFob: Message format for batch mode](#)
- [-WmsgFoi: Message format for interactive mode](#)
- [-WmsgFonf: Message format for no file information](#)

## 6.3.55 -WmsgNe: Number of error messages

### Group

### Messages

## Scope

Assembly Unit

## Syntax

```
-WmsgNe<number>
```

## Arguments

<number>: Maximum number of error messages.

## Default

50

## Description

With this option the amount of error messages can be reported until the Assembler stops assembling. Note that subsequent error messages which depends on a previous one may be confusing.

## Example

```
ASMOPTIONS=-WmsgNe2
```

The Assembler stops assembling after two error messages.

## See also

Assembler options:

- [-WmsgNi: Number of Information messages](#)
- [-WmsgNw: Number of Warning messages](#)

### 6.3.56 -WmsgNi: Number of Information messages

## Group

Messages

## Scope

Assembly Unit

## Syntax

`-WmsgNi<number>`

## Arguments

`<number>`: Maximum number of information messages.

## Default

50

## Description

With this option the maximum number of information messages can be set.

## Example

```
ASMOPTIONS=-WmsgNi10
```

Only ten information messages are logged.

## See also

Assembler options:

- [-WmsgNe: Number of error messages](#)
- [-WmsgNw: Number of Warning messages](#)

## 6.3.57 -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

## Description

The application produces some messages which are not in the normal message categories (WARNING, INFORMATION, ERROR, or FATAL). With this option such messages can be disabled. The purpose for this option is to reduce the amount of messages and to simplify the error parsing of other tools:

- a: The application provides information about all included files. With this suboption this option can be disabled.
- b: With this suboption messages about reading files e.g., the files used as input can be disabled.
- c: Disables messages informing about generated files.
- d: At the end of the assembly, the application may provide information about statistics, e.g., code size, RAM/ROM usage, and so on. With this suboption this option can be disabled.
- e: With this option, informal messages (e.g., memory model, floating point format, etc.) can be disabled.

## NOTE

Depending on the application, not all suboptions may make sense. In this case they are just ignored for compatibility.

## Example

```
-WmsgNu=c
```

## 6.3.58 -WmsgNw: Number of Warning messages

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WmsgNw<number>
```

### Arguments

<number>: Maximum number of warning messages.

### Default

50

### Description

With this option the maximum number of warning messages can be set.

### Example

```
ASMOPTIONS=-WmsgNw15
```

Only 15 warning messages are logged.

### See also

Assembler options:

- [-WmsgNe: Number of error messages](#)
- [-WmsgNi: Number of Information messages](#)

## 6.3.59 -WmsgSd: Setting a message to disable

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WmsgSd<number>
```

### Arguments

<number>: Message number to be disabled, e.g., 1801

### Default

None

### Description

With this option a message can be disabled so it does not appear in the error output.

### Example

```
-WmsgSd1801
```

### See also

Assembler options:

- [-WmsgSe: Setting a message to Error](#)
- [-WmsgSi: Setting a message to Information](#)
- [-WmsgSw: Setting a Message to Warning](#)

## 6.3.60 -WmsgSe: Setting a message to Error

### Group

Messages

### Scope

Assembly Unit

## Syntax

`-WmsgSe<number>`

## Arguments

`<number>`: Message number to be an error, e.g., 1853

## Default

None

## Description

Allows changing a message to an error message.

## Example

`-WmsgSe1853`

## See also

- [-WmsgSd: Setting a message to disable](#)
- [-WmsgSi: Setting a message to Information](#)
- [-WmsgSw: Setting a Message to Warning](#)

## 6.3.61 -WmsgSi: Setting a message to Information

### Group

Messages

### Scope

Assembly Unit

### Syntax

`-WmsgSi<number>`

### Arguments



<number>: Message number to be an information, e.g., 1853

## Default

None

## Description

With this option a message can be set to an information message.

## Example

```
-WmsgSi1853
```

## See also

Assembler options:

- [-WmsgSd: Setting a message to disable](#)
- [-WmsgSe: Setting a message to Error](#)
- [-WmsgSw: Setting a Message to Warning](#)

## 6.3.62 -WmsgSw: Setting a Message to Warning

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WmsgSw<number>
```

### Arguments

<number>: Error number to be a warning, e.g., 2901

### Default

None

### Description

With this option a message can be set to a warning message.

## Example

```
-WmsgSw2901
```

## See also

Assembler options:

- [-WmsgSd](#): Setting a message to disable
- [-WmsgSe](#): Setting a message to Error
- [-WmsgSi](#): Setting a message to Information

## 6.3.63 -WOutFile: Create error listing file

### Group

Messages

### Scope

Assembly Unit

### Syntax

```
-WOutFile (On|Off)
```

### Arguments

None

### Default

Error listing file is created.

### Description

This option controls if a error listing file should be created at all. The error listing file contains a list of all messages and errors which are created during a assembly process. Since the text error feedback can now also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the listing file is controlled by the environment variable [ERRORFILE: Filename specification error](#).

### Example

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014

-WOutFileOn

The error file is created as specified with `ERRORFILE`.

-WErrFileOff

No error file is created.

### See also

Assembler options:

- [-WErrFile: Create "err.log" error file](#)
- [-WStdout: Write to standard output](#)

## 6.3.64 -WStdout: Write to standard output

### Group

Messages

### Scope

Assembly Unit

### Syntax

-WStdout (On|Off)

### Arguments

None

### Default

output is written to `stdout`

### Description

With Windows applications, the usual standard streams are available. But text written into them does not appear anywhere unless explicitly requested by the calling application. With this option it can be controlled if the text to error file should also be written into `stdout`.

## Example

```
-WStdoutOn
```

All messages are written to `stdout`.

```
-WErrFileOff
```

Nothing is written to `stdout`.

## See also

Assembler options:

- [-WErrFile](#): Create "err.log" error file
- [-WOutFile](#): Create error listing file

## Chapter 7

# Sections

Sections are portions of code or data that cannot be split into smaller elements. Each section has a name, a type, and some attributes.

Each assembly source file contains at least one section. The number of sections in an assembly source file is only limited by the amount of memory available on the system at assembly time. If several sections with the same name are detected inside of a single source file, the code is concatenated into one large section.

Sections from different modules, but with the same name, will be combined into a single section at linking time.

Sections are defined through [Section attributes](#) and [Section types](#). The last part of the chapter deals with the merits of using relocatable sections. (See [Relocatable vs. absolute sections](#))

### 7.1 Section attributes

An attribute is associated with each section according to its content. A section may be:

- a data section,
- a constant data section, or
- a code section.

#### 7.1.1 Code sections

A section containing at least one instruction is considered to be a code section. Code sections are always allocated in the target processor's ROM area.

Code sections should not contain any variable definitions (variables defined using the `DS` directive). You do not have any write access on variables defined in a code section. In addition, variables in code sections cannot be displayed in the debugger as data.

## 7.1.2 Constant sections

A section containing only constant data definition (variables defined using the `DC` or `DCB` directives) is considered to be a constant section. Constant sections should be allocated in the target processor's `ROM` area, otherwise they cannot be initialized at application loading time.

## 7.1.3 Data sections

A section containing only variables (variables defined using the `DS` directive) is considered to be a data section. Data sections are always allocated in the target processor's `RAM` area.

### NOTE

A section containing variables (`DS`) and constants (`DC`) or code is not a data section. The default for such a section with mixed `DC` and code content is to put that content into `ROM`.

We strongly recommend that you use separate sections for the definition of variables and constant variables. This will prevent problems in the initialization of constant variables.

## 7.2 Section types

First of all, you should decide whether to use relocatable or absolute code in your application. The Assembler allows the mixing of absolute and relocatable sections in a single application and also in a single source file. The main difference between absolute and relocatable sections is the way symbol addresses are determined.

This section covers these two types of sections:

- [Absolute sections](#)
- [Relocatable sections](#)

## 7.2.1 Absolute sections

The starting address of an absolute section is known at assembly time. An absolute section is defined through the **ORG - Set Location Counter** assembler directive. The operand specified in the ORG directive determines the start address of the absolute section. The following listing shows an example of constructing absolute sections using the ORG assembler directive.

### Listing: Example source code using ORG for absolute sections

```
XDEF  entry
ORG   $8000 ; Absolute constant data section.

cst1: DC.B  $26
cst2: DC.B  $BC
...

      ORG   $080  ; Absolute data section.
var:  DS.B  1

      ORG   $8010 ; Absolute code section.

entry:
      LDA  cst1  ; Loads value in cst1
      ADD  cst2  ; Adds value in cst2
      STA  var   ; Stores result into var
      BRA  entry
```

In the previous example, two bytes of storage are allocated starting at address \$A00. The *constantvariable* - *cst1* - will be allocated one byte at address \$8000 and another constant - *cst2* - will be allocated one byte at address \$8001. All subsequent instructions or data allocation directives will be located in this absolute section until another section is specified using the ORG or SECTION directives.

When using absolute sections, it is the user's responsibility to ensure that there is no overlap between the different absolute sections defined in the application. In the previous example, the programmer should ensure that the size of the section starting at address \$8000 is not bigger than \$10 bytes, otherwise the section starting at \$8000 and the section starting at \$8010 will overlap.

Even applications containing only absolute sections must be linked. In that case, there should not be any overlap between the address ranges from the absolute sections defined in the assembly file and the address ranges defined in the linker parameter (PRM) file.

## Section types

The PRM file used to link the example above, can be defined as the following listing displays.

### Listing: Example PRM file for linking source code using ORG for absolute sections

```
LINK test.abs /* Name of the executable file generated. */
NAMES test.o /* Name of the object file in the application */

END

SECTIONS

/* READ_ONLY memory area. There should be no overlap between this
memory area and the absolute sections defined in the assembly
source file. */

MY_ROM = READ_ONLY 0x8000 TO 0xFDFE;

/* READ_WRITE memory area. There should be no overlap between this
memory area and the absolute sections defined in the assembly
source file. */

MY_RAM = READ_WRITE 0x0100 TO 0x023F;

END

PLACEMENT

/* Relocatable variable sections are allocated in MY_RAM. */

DEFAULT_RAM, SSTACK INTO MY_RAM;

/* Relocatable code and constant sections are allocated in MY_ROM. */

DEFAULT_ROM INTO MY_ROM;

END

STACKSTOP $014F /* Initializes the stack pointer */

INIT entry /* entry is the entry point to the application. */

VECTOR ADDRESS 0xFFFFE entry /* Initialization for Reset vector.*/
```

The linker PRM file contains at least:

- The name of the absolute file (`LINK` command).
- The name of the object file which should be linked (`NAMES` command).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined `DEFAULT_RAM` (or its ELF alias ``.data'`) section must be placed there. For applications containing only absolute sections, nothing will be allocated (`SECTIONS` and `PLACEMENT` commands).
- The specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section `DEFAULT_ROM` (or its ELF alias ``.data'`) must be placed there. For applications containing only absolute sections, nothing will be allocated (`SECTIONS` and `PLACEMENT` commands).



- The specification of the application entry point (`INIT` command)
- The definition of the reset vector (`VECTOR ADDRESS` command)

## 7.2.2 Relocatable sections

The starting address of a relocatable section is evaluated at linking time according to the information stored in the linker parameter file. A relocatable section is defined through the **SECTION - Declare Relocatable Section** assembler directive. The following listing shows an example using the `SECTION` directive.

### Listing: Example source code using SECTION for relocatable sections

```
        XDEF  entry
constSec: SECTION      ; Relocatable constant data section.

cst1:   DC.B  $A6
cst2:   DC.B  $BC

dataSec: SECTION      ; Relocatable data section.

var:    DS.B  1

codeSec: SECTION      ; Relocatable code section.

entry:

        LDA  cst1  ; Load value into cst1
        ADD  cst2  ; Add value in cst2
        STA  var   ; Store into var
        BRA  entry
```

In the previous example, two bytes of storage are allocated in the `constSec` section. The constant `cst1` is allocated at the start of the section at address `$A00` and another constant `cst2` is allocated at an offset of 1 byte from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable `constSec` section until another section is specified using the `ORG` or `SECTION` directives.

When using relocatable sections, the user does not need to care about overlapping sections. The linker will assign a start address to each section according to the input from the linker parameter file.

The user can decide to define only one memory area for the code and constant sections and another one for the variable sections or to split the sections over several memory areas.

## 7.2.2.1 Example: Defining one RAM and one ROM area.

When all constant and code sections as well as data sections can be allocated consecutively, the PRM file used to assemble the example above can be defined as the following listing displays.

### Listing: PRM file for defining one RAM area and one ROM area

```
LINK test.abs/* Name of the executable file generated. */
NAMES test.o /* Name of the object file in the application */

END

SECTIONS

/* READ_ONLY memory area. */
    MY_ROM = READ_ONLY 0x8000 TO 0xFDFE;

/* READ_WRITE memory area. */
    MY_RAM = READ_WRITE 0x0100 TO 0x023F;

END

PLACEMENT

/* Relocatable variable sections are allocated in MY_RAM. */
    DEFAULT_RAM, dataSec , SSTACK INTO MY_RAM;

/* Relocatable code and constant sections are allocated in MY_ROM. */
    DEFAULT_ROM, constSec INTO MY_ROM;

END

INIT entry /* entry is the entry point to the application. */
VECTOR ADDRESS 0xFFFFE entry /* Initialization for Reset vector.*/
```

The linker PRM file contains at least:

- The name of the absolute file (`LINK` command).
- The name of the object files which should be linked (`NAMES` command).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined `DEFAULT_RAM` section (or its ELF alias `.data`) must be placed there (`SECTIONS` and `PLACEMENT` commands).
- The specification of a memory area where the sections containing code or constants must be allocated. At least, the predefined `DEFAULT_ROM` section (or its ELF alias `.text`) must be placed there (`SECTIONS` and `PLACEMENT` commands).
- Constants sections should be defined in the ROM memory area in the `PLACEMENT` section (otherwise, they are allocated in RAM).
- The specification of the application entry point (`INIT` command).
- The definition of the reset vector (`VECTOR ADDRESS` command).

According to the PRM file listed above:

- the `dataSec` section will be allocated starting at `0x0080`.
- the `codeSec` section will be allocated starting at `0x0B00`.
- the `constSec` section will be allocated next to the `codeSec` section.

### 7.2.2.2 Example: Defining multiple RAM and ROM areas

When all constant and code sections as well as data sections cannot be allocated consecutively, the PRM file used to link the example above can be defined as the following listing displays:

#### Listing: PRM file for defining multiple RAM and ROM areas

```
LINK test.abs /* Name of the executable file generated. */
NAMES
    test.o /* Name of the object file in the application. */
END
SECTIONS
    /* Two READ_ONLY memory areas */
    ROM_AREA_1= READ_ONLY 0x8000 TO 0x800F;
    ROM_AREA_2= READ_ONLY 0x8010 TO 0xFDFD;
    /* Three READ_WRITE memory areas */
    RAM_AREA_1= READ_WRITE 0x0040 TO 0x00FF; /* zero-page memory area */
    RAM_AREA_2= READ_WRITE 0x0100 TO 0x01FF;
    MY_STK = READ_WRITE 0x0200 TO 0x023F; /* Stack memory area */
END
PLACEMENT
    /* Relocatable variable sections are allocated in MY_RAM. */
    dataSec INTO RAM_AREA_2;
    DEFAULT_RAM INTO RAM_AREA_1;
    SSTACK INTO MY_STK; /* Stack allocated in MY_STK */
    /* Relocatable code and constant sections are allocated in MY_ROM. */
    constSec INTO ROM_AREA_2;
    codeSec, DEFAULT_ROM INTO ROM_AREA_1;
END
INIT entry /* Application's entry point. */
```

```
VECTOR 0 entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

- The name of the absolute file (`LINK` command).
- The name of the object files which should be linked (`NAMES` command).
- The specification of memory areas where the sections containing variables must be allocated. At least, the predefined `DEFAULT_RAM` section (or its ELF alias ``.data'`)` must be placed there (`SECTIONS` and `PLACEMENT` commands).
- The specification of memory areas where the sections containing code or constants must be allocated. At least the predefined `DEFAULT_ROM` section (or its ELF alias ``.text'`)` must be placed there (`SECTIONS` and `PLACEMENT` commands).
- Constants sections should be defined in the ROM memory area in the `PLACEMENT` section (otherwise, they are allocated in RAM).
- The specification of the application entry point (`INIT` command)
- The definition of the reset vector (`VECTOR` command)

According to the PRM file listed above:

- the `dataSec` section is allocated starting at `0x0100`.
- the `constSec` section is allocated starting at `0x8000`.
- the `codeSec` section is allocated starting at `0x8010`.
- 64 bytes of RAM are allocated in the stack starting at `0x0200`.

## 7.3 Relocatable vs. absolute sections

Generally, we recommend developing applications using relocatable sections. Relocatable sections offer several advantages.

### 7.3.1 Modularity

An application is more modular when programming can be divided into smaller units called sections. The sections themselves can be distributed among different source files.

### 7.3.2 Multiple developers

When an application is split over different files, multiple developers can be involved in the development of the application. To avoid major problems when merging the different files, attention must be paid to the following items:

- An include file must be available for each assembly source file, containing `XREF` directives for each exported variable, constant and function. In addition, the interface to the function should be described there (parameter passing rules as well as the function return value).
- When accessing variables, constants, or functions from another module, the corresponding include file must be included.
- Variables or constants defined by another developer must always be referenced by their names.
- Before invoking a function implemented in another file, the developer should respect the function interface, i.e., the parameters are passed as expected and the return value is retrieved correctly.

### 7.3.3 Early development

The application can be developed before the application memory map is known. Often the application's definitive memory map can only be determined once the size required for code and data can be evaluated. The size required for code or data can only be quantified once the major part of the application is implemented. When absolute sections are used, defining the definitive memory map is an iterative process of mapping and remapping the code. The assembly files must be edited, assembled, and linked several times. When relocatable sections are used, this can be achieved by editing the PRM file and linking the application.

### 7.3.4 Enhanced portability

As the memory map is not the same for each derivative (MCU), using relocatable sections allow easy porting of the code for another MCU. When porting relocatable code to another target you only need to link the application again with the appropriate memory map.

### 7.3.5 Tracking overlaps

When using absolute sections, the programmer must ensure that there is no overlap between the sections. When using relocatable sections, the programmer does not need to be concerned about any section overlapping another. The labels' offsets are all evaluated relatively to the beginning of the section. Absolute addresses are determined and assigned by the linker.

### **7.3.6 Reusability**

When using relocatable sections, code implemented to handle a specific I/O device (serial communication device), can be reused in another application without any modification.

## Chapter 8

# Assembler Syntax

An assembler source program is a sequence of source statements. Each source statement is coded on one line of text and can be either a:

- [Comment line](#) or a
- [Source line](#).

### 8.1 Comment line

A comment can occupy an entire line to explain the purpose and usage of a block of statements or to describe an algorithm. A comment line contains a semicolon followed by a text, as listed the following listing. Comments are included in the assembly listing, but are not significant to the Assembler.

An empty line is also considered to be a comment line.

#### Listing: Examples of comments

```
; This is a comment line followed by an empty line and non comments  
... (non comments)
```

### 8.2 Source line

Each source statement includes one or more of the following four fields:

- a [Label field](#),
- an [Operation field](#),
- one or several operands, or
- a comment.

Characters on the source line may be either upper or lower case. Directives and instructions are case-insensitive, whereas symbols are case-sensitive unless the assembler option for case insensitivity on label names ( **-Ci: Switch case sensitivity on label names OFF**) is activated.

## 8.2.1 Label field

The label field is the first field in a source line. A label is a symbol followed by a colon. Labels can include letters (A-Z or a-z), underscores, periods and numbers. The first character must not be a number.

### NOTE

For compatibility with other Assembler vendors, an identifier starting on column 1 is considered to be a label, even when it is not terminated by a colon. When the **-MCUasm: Switch compatibility with MCUasm ON** assembler option is activated, you *MUST* terminate labels with a colon. The Assembler produces an error message when a label is not followed by a colon.

Labels are required on assembler directives that define the value of a symbol (`SET` or `EQU`). For these directives, labels are assigned the value corresponding to the expression in the operand field.

Labels specified in front of another directive, instruction or comment are assigned the value of the location counter in the current section.

### NOTE

When the Macro Assembler expands a macro it generates internal symbols starting with an underscore `_`. Therefore, to avoid potential conflicts, user defined symbols should not begin with an underscore

### NOTE

For the Macro Assembler, a `.B` or `.W` at the end of a label has a specific meaning. Therefore, to avoid potential conflicts, user-defined symbols should not end with `.B` or `.W`.

## 8.2.2 Operation field



The operation field follows the label field and is separated from it by a white space. The operation field must not begin in the first column. An entry in the operation field is one of the following:

- an instruction's mnemonic - an abbreviated, case-insensitive name for a member in the [Instruction set](#)
- a [Directive](#) name, or
- a [Macro](#) name.

## 8.2.2.1 Instruction set

Executable instructions for the M68HC08 processor are defined in the *CPU08 Reference Manual*.

### 8.2.2.1.1 HC08 instruction set

The following table presents an overview of the instructions available for the HC08:

**Table 8-1. HC08 instruction set**

Instruction	Addressing modes	Descriptions
ADC	#<expression> <expression> <expression>,X ,X <expression>,SP	Add with Carry
ADD	#<expression> <expression> <expression>,X ,X <expression>,SP	Add without carry
AIS	#<expression>	Add Immediate value (signed) to Stack Pointer
AIX	#<expression>	Add Immediate value (signed) to Index register H:X
AND	#<expression> <expression> <expression>,X ,X <expression>,SP	Logical AND
ASL	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Left
ASLA		Arithmetic Shift Left Accumulator
ASLX		Arithmetic Shift Left register X
ASR	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Right
ASRA		Arithmetic Shift Right Accumulator
ASRX		Arithmetic Shift Right register X
BCC	<label>	Branch if Carry bit Clear
BCLR	BitNumber, <expression>	Clear one Bit in memory
BCS	<label>	Branch if Carry bit Set

Table continues on the next page...

**Table 8-1. HC08 instruction set (continued)**

Instruction	Addressing modes	Descriptions
BEQ	<label>	Branch if Equal
BGE	<label>	Branch if Greater Than or Equal to
BGND		Enter Background Debug Mode. Only available for HCS08 (-CS08 option)
BGT	<label>	Branch if Greater Than
BHCC	<label>	Branch if Half Carry bit Clear
BHCS	<label>	Branch if Half Carry bit Set
BHI	<label>	Branch if Higher
BHS	<label>	Branch if Higher or Same
BIH	<label>	Branch if /IRQ Pin High
BIL	<label>	Branch if /IRQ Pin Low
BIT	#<expression> <expression> <expression>,X ,X <expression>,SP	Bit Test
BLE	<label>	Branch if Less Than or Equal To
BLO	<label>	Branch if Lower (same as BCS)
BLS	<label>	Branch if Lower or Same
BLT	<label>	Branch if Less Than
BMC	<label>	Branch if interrupt Mask Clear
BMI	<label>	Branch if Minus
BMS	<label>	Branch If interrupt Mask Set
BNE	<label>	Branch if Not Equal
BPL	<label>	Branch if Plus
BRA	<label>	Branch Always
BRCLR	BitNumber, <expression>, <label>	Branch if Bit is Clear
BRN	<label>	Branch Never
BRSET	BitNumber, <expression>, <label>	Branch if Bit Set
BSET	BitNumber,<expression>	Set Bit in memory
BSR	<label>	Branch to Subroutine
CBEQ	<expression>,<label> <expression>,X +,<label> X+,<label> <expression>,SP,<label>	Compare and Branch if Equal
CBEQA	#<expression>,<label>	
CBEQX	#<expression>,<label>	
CLC		Clear Carry bit
CLI		Clear Interrupt mask bit
CLR	<expression> <expression>,X ,X <expression>,SP	Clear memory
CLRA		Clear Accumulator A
CLRH		Clear index Register H
CLR X		Clear index Register X

Table continues on the next page...

**Table 8-1. HC08 instruction set (continued)**

Instruction	Addressing modes	Descriptions
CMP	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare accumulator with memory
COM	<expression> <expression>,X ,X <expression>,SP	One's complement on memory location
COMA		One's complement on accumulator A
COMX		One's complement on register X
CPHX	#<expression> <expression> <expression>,SP	Compare index register H:X with memory Stack pointer and Extended addressing modes only available for HCS08 (-CS08 option)
CPX	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare index register X with memory
DAA		Decimal Adjust Accumulator
DBNZ	<expression>,<label> <expression>,X,<label> X,<label> <expression>,SP,<label>	Decrement counter and Branch if Not Zero
DBNZA	<label>	
DBNZX	<label>	
DEC	<expression> <expression>,X ,X <expression>,SP	Decrement memory location
DECA		Decrement Accumulator
DECX		Decrement Index register
DIV		Divide
EOR	#<expression> <expression> <expression>,X ,X <expression>,SP	Exclusive OR Memory with accumulator
INC	<expression> ,X <expression>,X <expression>,SP	Increment memory location
INCA		Increment Accumulator
INCX		Increment register X
JMP	<expression> <expression>,X ,X	Jump to label
JSR	<expression> <expression>,X ,X	Jump to Subroutine
LDA	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Accumulator
LDHX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Index register H:X from memory Indexed, Stack pointer and extended addressing modes are only available for HCS08 (-CS08 option).
LDX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load index Register X from memory
LSL	<expression> <expression>,X ,X <expression>,SP	Logical Shift Left in memory
LSLA		Logical Shift Left Accumulator
LSLX		Logical Shift Left register X
LSR	<expression> <expression>,X ,X <expression>,SP	Logical Shift Right in memory

Table continues on the next page...

**Table 8-1. HC08 instruction set (continued)**

Instruction	Addressing modes	Descriptions
LSRA		Logical Shift Right Accumulator
LSRX		Logical Shift Right register X
MOV	<expression>,<expression> <expression>,X+ #<expression>,<expression> X +,<expression>	Memory-to-memory byte Move
MUL		Unsigned multiply
NEG	<expression> <expression>,X ,X <expression>,SP	Two's complement in memory
NEGA		Two's complement on Accumulator
NEGX		Two's complement on register X
NOP		No operation
NSA		Nibble Swap Accumulator
ORA	#<expression> <expression> <expression>,X ,X <expression>,SP	Inclusive OR between Accumulator and memory
PSHA		Push Accumulator onto stack
PSHH		Push index register H onto stack
PSHX		Push index register X onto stack
PULA		Pull Accumulator from stack
PULH		Pull index register H from stack
PULX		Pull index register X from stack
ROL	<expression> <expression>,X ,X <expression>,SP	Rotate memory Left
ROLA		Rotate Accumulator Left
ROLX		Rotate register X Left
ROR	<expression> <expression>,X ,X <expression>,SP	Rotate memory Right
RORA		Rotate Accumulator Right
RORX		Rotate register X Right
RSP		Reset Stack Pointer
RTI		Return from Interrupt
RTS		Return from Subroutine
SBC	#<expression> <expression> <expression>,X ,X <expression>,SP	Subtract with Carry
SEC		Set Carry bit
SEI		Set Interrupt mask bit
STA	<expression> <expression>,X ,X <expression>,SP	Store Accumulator in Memory
STHX	<expression> <expression>,SP	Store Index register H:X Stack pointer and extended addressing modes are only available for HCS08 (-CS08 option)
STOP		Enable IRQ pin and Stop oscillator

Table continues on the next page...

**Table 8-1. HC08 instruction set (continued)**

Instruction	Addressing modes	Descriptions
STX	<expression> <expression>,X ,X <expression>,SP	Store index register X in memory
SUB	#<expression> <expression> <expression>,X ,X <expression>,SP	Subtract
SWI		Software Interrupt
TAP		Transfer Accumulator to CCR
TAX		Transfer Accumulator to index Register X
TPA		Transfer CCR to Accumulator
TST	<expression> <expression>,X ,X <expression>,SP	Test memory for negative or zero
TSTA		Test Accumulator for negative or zero
TSTX		Test register X for negative or zero
TSX		Transfer SP to index register H:X
TXA		Transfer index register X to Accumulator
TXS		Transfer index register X to SP
WAIT		Enable interrupts; stop processor

### 8.2.2.1.2 Special HCS08 instructions

The following table lists the instructions which HCS08 core provides in addition to the HC08 core instructions:

**Table 8-2. Special HC(S)08 instructions**

Instruction	Addressing modes	Descriptions
BGND		Enter Background Debug Mode. Only available with the <a href="#">-CS08/-C08/-CRS08: Derivative family</a> assembler options.
CPHX	#<expression> <expression> <expression>,SP	Compare index register H:X with memory Stack pointer and extended addressing modes are only available with the -CS08, -C08, or -CRS08 assembler options.
LDHX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load index register H:X from memory Indexed, stack pointer, and extended addressing modes are only available with the -CS08 option
STHX	<expression> <expression>,SP	Store index register H:X Stack pointer and extended addressing modes are only available with the -CS08option.

### 8.2.2.1.3 RS08 instruction set

The following table presents an overview of the instructions available for the RS08.

**Table 8-3. RS08 instructions set**

Instruction	Addressing Modes	Description
ADC	#<expression> <expression> ,X D[X] X	Add with Carry
ADCX		Alias for ADC X
ADD	#<expression> <expression> ,X D[X] X	Add without Carry
ADDX		Alias for ADD X
AND	#<expression> <expression> ,X D[X] X	Logical AND
ANDX		Alias for AND X
ASLA		Arithmetic Shift Left Accumulator (alias for LSLA)
BCC	<label>	Branch if Carry Bit Clear
BCLR	BitNumber, <expression> BitNumber,D[X] BitNumber,X	Clear one Bit in Memory
BCS	<label>	Branch if Carry Bit Set
BEQ	<label>	Branch if Equal
BGND		Background
BHS	<label>	Branch if Higher or Same
BLO	<label>	Branch if Lower
BNE	<label>	Branch if Not Equal
BRN	<label>	Branch Never (Alias for BRA *+\$2)
BRCLR	BitNumber, <expression>, <label> BitNumber,D[X],<label> BitNumber,X,<label>	Branch if Bit is Clear
BRSET	BitNumber, <expression>, <label> BitNumber,D[X],<label> BitNumber,X,<label>	Branch if Bit Set
BSET	BitNumber,<expression> BitNumber,D[X] BitNumber,X	Set Bit in Memory
BSR	<label>	Branch to Subroutine
CBEQ	<expression>,<label> #<expression>,<label> ,X,<label> D[X],<label> X,<label>	Compare and Branch if Equal
CBEQA	<label>	
CBEQX	<label>	
CLC		Clear Carry Bit
CLR	<expression> ,X D[X] X	Clear Memory
CLR X		Clear Index Register X
CMP	#<expression> <expression> ,X D[X] X	Compare Accumulator with Memory
COMA		Complement (One's Complement)

Table continues on the next page...

**Table 8-3. RS08 instructions set (continued)**

Instruction	Addressing Modes	Description
DBNZ	<expression>, <label> ,X, <label> D[X], <label> X, <label>	Decrement Counter and Branch if Not Zero
DBNZA	<label>	
DBNZX	<label>	
DEC	<expression> ,X D[X] X	Decrement Memory Location
DEC	<\$13	Force tiny addressing (will use \$03)
DECA		Decrement Accumulator
DECX		Decrement Index Register
EOR	#<expression> <expression> D[X] ,X X	Exclusive OR Memory with Accumulator
EORX		Exclusive OR (index register and accumulator)
INC	<expression> ,X D[X] X	Increment Memory Location
INC	>\$01	Force direct addressing
INCA		Increment Accumulator
INCX		Increment Register X
JMP	<label>	Jump to Label
JSR	<label>	Jump to Subroutine
LDA	#<expression> <expression> ,X D[X] X	Load Accumulator indexed
LDA	<\$0FF	Force short addressing (will use \$1F)
LDX	#<expression> <expression> ,X D[X] X	Load Index Register X from Memory
LDX	\$0FF	Load Direct
LSLA		Logical Shift Left Accumulator
LSRA		Logical Shift Right Accumulator
MOV	<expression>, <expression> #<expression>, <expression> D[X], <expression> <expression>, D[X] #<expression>, D[X]	Memory to Memory Byte Move
NOP		No Operation
ORA	#<expression> <expression> ,X D[X] X	Inclusive OR between Accumulator and Memory
ORAX		Inclusive OR between Accumulator and Index Register
ROLA		Rotate Accumulator Left
RORA		Rotate Accumulator Right
RTS		Return from Subroutine
SBC	#<expression> <expression> ,X D[X] X	Subtract with Carry
SBCX		Subtract with Carry (Index Register content from Accumulator)
SEC		Set Carry Bit
SHA		Swap Shadow PC High with A
SLA		Swap Shadow PC Low with A
STA	<expression> ,X D[X] X	Store Accumulator in Memory

Table continues on the next page...

**Table 8-3. RS08 instructions set (continued)**

Instruction	Addressing Modes	Description
STOP		Stop Processing
STX	<expression>	Store Index Register X in Memory
SUB	#<expression> <expression> ,X D[X]	Subtract
SUBX		
TAX		Transfer Accumulator to Index Register X
TST	#<expression> <expression> ,X D[X]	Test for zero (alias for MOV <expression>, <expression>)
TSTA		Test Accumulator (alias for ORA #0)
TSTX		Test Index Register X (alias for MOV X,X)
TXA		Transfer Index Register X to Accumulator
WAIT		Enable Interrupts; Stop Processor

### NOTE

For RS08 both D[X] and ,X notations refer to the memory location \$000E. The ,X notation is supported for compatibility reasons with HC(S)08. Wherever ,X is supported, D[X] is also supported. In situations where the use of ,X would lead to double commas (e.g. BCLR 0,,X) the use of ,X is not allowed.

#### 8.2.2.2 Directive

Assembler directives are described in the [Assembler Directives](#) chapter of this manual.

#### 8.2.2.3 Macro

A user-defined macro can be invoked in the assembler source program. This results in the expansion of the code defined in the macro. Defining and using macros are described in the [Macros](#) chapter in this manual.

### 8.2.3 Operand field: Addressing modes (HC(S)08)



The operand fields, when present, follow the operation field and are separated from it by a white space. When two or more operand subfields appear within a statement, a comma must separate them.

The following table lists the addressing mode notations allowed in the operand field:

**Table 8-4. HC(S)08 addressing mode notation**

Addressing Mode	Notation	Example
Inherent	No operands	RSP
Immediate	#<expression>	ADC #\$01
Direct	<expression>	ADC byte
Extended	<expression>	ADC word
Indexed, no offset	,X	ADC ,X
Indexed, 8-bit offset	<expression>,X	ADC Offset,X
Indexed, 16-bit offset	<expression>,X	ADC Offset,X
Relative	<label>	BRA Label
Stack Pointer, 8-bit offset	<expression>,SP	ADC Offset,SP
Stack Pointer, 16-bit offset	<expression>,SP	ADC Offset,SP
Memory-to-memory immediate-to-direct	#<expression>,<expression>	MOV #\$05,MyDataByte
Memory-to-memory direct-to-direct	<expression>,<expression>	MOV DatLoc1,DatLoc2
Memory-to-memory indexed-to-direct with post-increment	X+,<expression>	MOV X+,<expression>
Memory-to-memory direct-to-indexed with post-increment	<expression>,X+	MOV <expression>,X+
Indexed with post-increment	X+	CBEQ X+, Data
Indexed, 8-bit offset, with post-increment	#<expression>,X+	CBEQ #offset,X+,Data

### 8.2.3.1 Inherent

Instructions using this addressing mode do not have any associated instruction fetch. Some of them are acting on data in the CPU registers. The following listing shows the inherent addressing-mode instructions.

#### Listing: Inherent addressing-mode instructions

```
CLRA
DAA
```

### 8.2.3.2 Immediate

#### Source line

The opcode contains the value to use with the instruction rather than the address of this value.

The effective address of the instruction is specified using the # character, as listed in the following listing:

#### Listing: Immediate addressing mode

```

        XDEF  Entry
initStack: EQU  $0400

MyData:  SECTION

data:    DS.B  1

MyCode:  SECTION

Entry:

        LDHX #initStack ; init Stack Pointer

        TXS                ; with value $400-1 = $03FF

main:    LDA  #100          ; load register A with (decimal) 100

        BRA  main

```

In this example, the hexadecimal value \$0400 is loaded in value in the register HX and the decimal value 100 is loaded into register A.

### 8.2.3.3 Direct

The direct addressing mode is used to address operands in the direct page of the memory (location \$0000 to \$00FF).

For most of the direct instructions, only two bytes are required: the first byte is the opcode and the second byte is the operand address located in page zero. See the following listing for an example of the direct addressing mode.

#### Listing: Direct addressing mode

```

        XDEF  Entry
initStack: EQU  $0400

MyData:  SECTION SHORT

data:    DS.B  1

MyCode:  SECTION

Entry:

        LDHX #initStack ; init Stack Pointer

        TXS                ; with value $400 - 1 = $03FF

```

```
main:      LDA    #55
           STA    data
           BRA    main
```

In this example, the value `55` is stored in the variable `data`, which is located on the direct page. The `MyData` section must be defined in the direct page in the linker parameter file. The opcode generated for the `STA data` instruction is two bytes long.

### 8.2.3.4 Extended

The extended addressing mode is used to access memory location located above the direct page in a 64-kilobyte memory map.

For the extended instructions, three bytes are required: the first byte is the opcode and the second and the third bytes are the most and least significant bytes of the operand address. See the following listing for an example of the extended addressing mode.

#### Listing: Extended addressing mode

```
          XDEF   Entry
initStack: EQU   $0400

          ORG   $B00

data:     DS.B   1
MyCode:   SECTION
Entry:

          LDHX  #initStack ; init Stack Pointer
          TXS                      ; with value $400-1 = $03FF

main:     LDA    #55
          STA    data
          BRA    main
```

In this example, the value `55` is stored in the variable `data`. This variable is located at address `$0B00` in the memory map. The opcode of the `STAdata` instruction is then three bytes long.

### 8.2.3.5 Indexed, no offset

#### Source line

This addressing mode is used to access data with variable addresses through the HX index register of the HC08 controller. The X index register contains the least significant byte of the operand while the H index register contains the most significant byte.

Indexed, no offset instructions are one byte long. See the following listing for an example of using the indexed (no offset) addressing mode.

#### Listing: Indexed (no offset) addressing mode

```

...
Entry:
    ...
    LDHX  #$0FFE
    LDA   ,X
    ...
    JMP   ,X
    ...

```

The value stored in memory location `$0FFE` is loaded into accumulator A. The `JMP` instruction causes the program to jump to the address pointed to by the HX register.

### 8.2.3.6 Indexed, 8-bit offset

This addressing mode is useful when selecting the k-th element in an n-element table. The size of the table is limited to 256 bytes.

Indexed, 8-bit offset instructions are two byte long. The first byte is the opcode and the second byte contains the index register offset byte. See the following listing for an example of using the indexed (8-bit offset) addressing mode.

#### Listing: Index (8-bit offset) addressing mode

```

        XDEF  Entry
initStack: EQU  $0400

MyData:  SECTION SHORT

data:    DS.B  8

MyCode:  SECTION

Entry:

        LDHX #initStack ; init Stack Pointer

        TXS                ; with value $400-1 = $03FF

main:

```

```

        LDHX  #data
        LDA   5 ,X
        ...
        JMP   $FF,X
    ...

```

The value contained in the memory at the location calculated using the address of data (pointed to by the HX register) + 5 is loaded in accumulator A. The `JMP` instruction causes the program to jump to the address pointed to by the HX register + `$FF`.

### 8.2.3.7 Indexed, 16-bit offset

This addressing mode is useful when selecting the k-th element in an n-element table. The size of the table is limited to `$FFFF` bytes.

Indexed, 16-bit offset instructions are three byte long. The first byte contains the opcode and the second and the third the high and low index register offset bytes. See the following listing for an example of using the indexed (16-bit offset) addressing mode.

#### Listing: Indexed (16-bit offset) addressing mode

```

        XDEF  Entry
initStack: EQU  $0400

MyData:   SECTION

data:     DS.B  8

MyCode:   SECTION

Entry:

        LDHX #initStack ; init Stack Pointer
        TXS          ; with value $400-1 = $03FF

main:

        LDHX #table
        STA  $500 ,X
        ...

J
MP  $1000,X
    ...

```

#### Source line

The value contained in the memory at the location calculated using the address of data (pointed to by register HX) + \$500 is loaded in accumulator A. The JMP instruction causes the program to jump to the address pointed to by the HX register + \$1000.

### 8.2.3.8 Relative

This addressing mode is used by all branch instructions to determine the destination address. The signed byte following the opcode is added to the contents of the program counter.

As the offset is coded on a signed byte, the branching range is -127 to +128. The destination address of the branch instruction must be in this range. See the following listing for an example of using the relative addressing mode.

#### Listing: Relative addressing mode

```
main:
    NOP
    NOP
    BRA    main
```

### 8.2.3.9 Stack Pointer, 8-bit offset

Stack Pointer, 8-bit offset instructions behave the same way than Indexed 8-bit offset instructions, except that the offset is added to the Stack Pointer SP in place of the HX Index register.

This addressing mode allow easy access of the data on the stack. If the interrupts are disabled, the Stack pointer can also be used as a second Index register. See the following listing for an example of using the Stack Pointer \*(8-bit offset) addressing mode.

#### Listing: Stack Pointer (8-bit offset) addressing mode

```
entry:
    LDHX  #$0500    ; init Stack Pointer to 04FF
    TXS
    LDA   #$40
    STA  $50, SP    ; Location $54F = $40
```

In this example, stack pointer, 8-bit offset mode is used to store the value \$40 in memory location \$54F.

### 8.2.3.10 Stack Pointer, 16-bit offset

Stack Pointer, 16-bit offset instructions behave the same way than Indexed, 16-bit offset instructions, except that the offset is added to the Stack Pointer (SP) in place of the HX Index register.

This addressing mode allow easy access of the data on the stack. If the interrupts are disabled, the Stack pointer can also be used as a second Index register. See the following listing for an example of using the Stack Pointer (16-bit offset) addressing mode.

#### Listing: Stack Pointer (16-bit offset) addressing mode

```
entry:
    LDHX  #$0100    ; init Stack Pointer to 00FF
    TXS
    LDA   $0500, SP ; Content of memory location $5FF is loaded in A
```

In this example, stack pointer, 16-bit offset mode is used to store the value in memory location \$5FF in accumulator A.

### 8.2.3.11 Memory-to-memory immediate-to-direct

This addressing mode is generally used to initialize variables and registers in page zero. The register A is not affected. See the following listing for an example for using the memory-to- memory immediate-to-direct addressing mode.

#### Listing: Memory-to-memory immediate-to-direct addressing mode

```
MyData:  EQU   $50
entry:
    MOV   #$20, MyData
```

The `MOV #$20, MyData` instruction stores the value \$20 in memory location \$50`MyData'.

### 8.2.3.12 Memory-to-memory direct-to-direct

#### Source line

This addressing mode is generally used to transfer variables and registers in page zero. The A register is not affected. See the following listing for an example of using the memory-to-memory direct-to-direct addressing mode.

#### Listing: Memory-to-memory direct-to-direct addressing mode

```
MyData1: EQU   $50
MyData2: EQU   $51

entry:

    MOV    #$10, MyData1
    MOV    MyData1, MyData2
```

The `MOV #$10,MyData1` instruction stores the value `$10` in memory location `$50`MyData1'` using the memory-to-memory Immediate-to-Direct addressing mode. The `MOV MyData1,MyData2` instruction moves the content of `MyData1` into `MyData2` using memory to memory Direct-to-Direct addressing mode. The content of `MyData2` (memory location `$51`) is then `$10`.

### 8.2.3.13 Memory-to-memory indexed-to-direct with post-increment

This addressing mode is generally used to transfer tables addressed by the index register to a register in page zero.

The operand addressed by the HX index register is stored in the direct page location addressed by the byte following the opcode. The HX index register is automatically incremented. The A register is not affected. See the following listing for an example of using the memory-to-memory indexed to direct with post-increment addressing mode.

#### Listing: Memory-to-memory indexed-to-direct with post increment addressing mode

```
                XDEF   Entry
ConstSCT: SECTION

Const:         DC.B   1,11,21,31,192,12,0

DataSCT: SECTION SHORT

MyReg:         DS.B   1

CodeSCT: SECTION

Entry:         LDHX   #$00FF

                TXS

main:

                LDHX   #Const

LOOP:          MOV    X+, MyReg
```



```

    BEQ    main
    BRA    LOOP
    
```

In this example, the table `Const` contains seven bytes defined in a constant section in ROM. The last value of this table is zero.

The HX register is initialized with the address of `Const`. All the values of this table are stored one after another in page-zero memory location `MyReg` using the `MOV X+, MyReg` instruction. When the value `0` is encountered, the HX register is reset with the address of the first element of the `#Const` table.

### 8.2.3.14 Memory-to-memory direct-to-indexed with post-increment

This addressing mode is generally used to fill tables addressed by the index register from registers in page zero.

The operand in the direct page location addressed by the byte following the opcode is stored in the memory location pointed to by the HX index register. The HX index register is automatically incremented. The A register is not affected. See the following listing for an example of using the memory-to-memory direct-to-indexed with post-increment addressing mode.

#### Listing: Memory-to-memory direct-to-indirect with post-increment addressing mode

```

MyData:    XDEF    entry
           SECTION SHORT

MyReg1:    DS.B    1

MyReg2:    DS.B    1

MyCode:    SECTION

entry:

           LDA    #$02

           STA    MyReg1

           INCA

           STA    MyReg2

           LDHX   #$1000

           MOV    MyReg1, X+

           MOV    MyReg2, X+

main:     BRA    main
    
```

#### Source line

The page-zero memory locations `MyReg1` and `MyReg2` are first respectively initialized with `$02` and `$03`. The contents of those data are then written in memory location `$1000` and `$1001`. The HX register points to memory location `$1002`.

### 8.2.3.15 Indexed with post-increment

The operand is addressed then the HX register is incremented.

This addressing mode is useful for searches in tables. It is only used with the `CBEQ` instruction. See the following listing for an example of an example of using the indexed with post-increment addressing mode.

#### Listing: Example of the indexed with post-increment addressing mode

```

XDEF Entry
ORG   $F000

data:  DC.B  1,11,21,31,$C0,12

CodeSCT: SECTION

Entry:  LDHX  # $00FF

      TXS

main:

      LDA  # $C0

      LDHX # data

LOOP:  CBEQ  X+, IS_EQUAL

      BRA  LOOP

IS_EQUAL: ...

```

Using this addressing mode, it is possible to scan the memory to find a location containing a specific value.

The value located at the memory location pointed to by HX is compared to the value in the A register. If the two values match, the program branches to `IS_EQUAL`. HX points to the memory location next to the one containing the searched value.

In this example, the value `$C0` is searched starting at memory location `$F000`. This value is found at the memory location `$F004`, the program branches to `IS_EQUAL`, and the HX register contains `$F005`.

### 8.2.3.16 Indexed, 8-bit offset, with post-increment

The address of the operand is the sum of the 8-bit offset added to the value in register HX.

The operand is addressed, then the HX register is incremented.

This addressing mode is useful for searches in tables. It is only used with the `CBEQ` instruction. See the following listing for an example of the indexed (8-bit offset) with post-increment addressing mode.

### Listing: Indexed (8-bit offset) with post-increment addressing mode

```

XDEF  Entry
ORG   $F000

data:  DCB.B $40,$00

       DC.B 1,11,21,31,$C0,12 ; $C0 is located at $F000+$40+4

CodeSCT: SECTION
Entry:  LDHX  # $00FF

       TXS

main:

       LDA  # $C0

       LDHX # data

LOOP:  CBEQ  $30,X+,IS_EQUAL

       BRA  LOOP

IS_EQUAL: ...

```

Using this addressing mode, it is possible to scan the memory to find a location containing a specific value starting at a specified location to which is added an offset.

The value located at memory location pointed to by  $HX + \$30$  is compared to the value in the A register. If the two values match, program branch to `IS_EQUAL`. HX points to memory location next to the one containing the searched value.

In this example, the value `$C0` is searched starting at memory location  $\$F000 + \$30 = \$F030$ . This value is found at memory location  $\$F044$ , the program branches to `IS_EQUAL`. The HX register contains the memory location of the searched value minus the offset, incremented by one:  $\$F044 - \$30 + 1 = \$F015$ .

## 8.2.4 Operand Field: Addressing Modes (RS08)

The following addressing mode notations are allowed in the operand field for the RS08:

**Table 8-5. Operand Field RS08 Addressing Modes**

Addressing Mode	Notation	Example
Inherent	No operands	RTS
Tiny	<expression>	ADD fourbits
Short	<expression>	CLR fivebits
Direct	<expression>	ADC byte
Extended	<expression>	JSR word
Relative	<label>	BRA Label
Immediate	#<expression>	ADC #\$01
Indexed	D[X] or ,X	ADC D[X] or ADC ,X

### 8.2.4.1 Inherent (RS08)

Instructions using this addressing mode have no associated instruction fetch. Some of them are acting on data in the CPU registers.

Example:

```
CLRA
INCA
NOP
```

### 8.2.4.2 Tiny

The tiny addressing mode is used to access only the first 16 bytes of the memory map (addresses from \$0000 to \$000F). The instructions using this addressing mode are encoded using one byte only. This addressing mode is available for INC, DEC, ADD and SUB instructions.

Example:

```
XDEF Entry
MyData: SECTION RS08_TINY
data: DS.B 1
MyCode: SECTION
```

```
Entry:
main:
ADD data

        BRA main
```

In this example, the value of the variable `data` is added to the accumulator. The data is located in the tiny memory area, so the encoding of the `ADD` instruction will be one byte long. Note that the tiny section has to be placed into the tiny memory area at link time.

### 8.2.4.3 Short

The RS08 short addressing mode is used to access only the first 32 bytes of the memory map (addresses from `$0000` to `$001F`). The instructions using this addressing mode are encoded using one byte only. This addressing mode is available for `CLR`, `LDA` and `STA` instructions.

Example:

```
        XDEF Entry
MyData: SECTION RS08_SHORT
data:   DS.B 1
MyCode: SECTION
Entry:
main:
LDA data

        BRA main
```

In this example, the value of the variable `data` is loaded into the accumulator. The data is located in the short memory area, so the encoding of the `LDA` instruction will be one byte long. Note that the short section has to be placed into the tiny memory area at linktime.

### 8.2.4.4 Direct

The direct addressing mode is used to address operands in the direct page of the memory (location `$0000` to `$00FF`).

## Source line

### Example:

```

XDEF Entry
MyData: SECTION
data: DS.B 1
MyCode: SECTION
Entry:
main: LDA #$55
STA data
      BRA main

```

In this example, the value `55` is stored in the variable `data`. The opcode generated for the instruction `STA data` is two bytes long.

### 8.2.4.5 Extended

The extended addressing mode is used only for `JSR` and `JMP` instructions. The 14-bit address is located in the lowest 14 bits of the encoding after the two-bit opcode.

### Example:

```

XDEF Entry
XREF target
data: DS.B 1
MyCode: SECTION
Entry:
main: LDA #$55

JMP target

```

In this example a jump is executed at an address defined by the external symbol `target`.

### 8.2.4.6 Relative

This addressing mode is used by all branch instructions to determine the destination address. The signed byte following the opcode is added to the contents of the program counter.

As the offset is coded on a signed byte, the branching range is `-127` to `+128`. The destination address of the branch instruction must be in this range.

### Example:

```

main:
NOP
NOP
BRA main

```

### 8.2.4.7 Immediate

The opcode contains the value to use with the instruction rather than the address of this value. The effective address of the instruction is specified using the # character as in the example below.

Example:

```

XDEF Entry
MyData: SECTION
data: DS.B 1
MyCode: SECTION

Entry:
main:
LDA #100
        BRA main

```

In this example, the decimal value 100 is loaded in register A.

### 8.2.4.8 Indexed

When using the indexed addressing mode, an index register is used as reference to access the instruction's operand. For the RS08, the index registers are located at \$000F (register X) and \$000E (register D[X]). The D[X] register is called the index data register, and can be designated by either one of the D[X] or, x notations. As a restriction, when the use of, x would lead to double commas in the assembly source, the use of, x is not allowed.

Example:

```

XDEF Entry

MyData: SECTION

data: DS.B 1

MyCode: SECTION

Entry:

main:
CLR D[X] ; equivalent to CLR ,X

CLR X

```

In this example the contents of both X and D[X] registers are replaced by zeros.

## 8.2.5 Comment Field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in the comment field.

Example:

```
    NOP  
; Comment following an instruction
```

## 8.3 Symbols

The following types of symbols are the topics of this section:

- [User-defined symbols](#)
- [External symbols](#)
- [Undefined symbols](#)
- [Reserved symbols](#)

### 8.3.1 User-defined symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

- The section, in which the memory location is defined
- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the [SECTION - Declare Relocatable Section](#) assembler directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line.

The following listing shows an example of a user-defined relocatable SECTION.

#### **Listing: Example of a user-defined relocatable SECTION**



```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec.

label2: DC.B 5 ; label2 is assigned offset 2 within Sec.

label3: DC.B 1 ; label3 is assigned offset 7 within Sec.
```

It is also possible to define a label with either an absolute or a previously defined relocatable value, using the [SET - Set Symbol Value](#) or [EQU - Equate symbol value](#) assembler directives.

Symbols with absolute values must be defined with constant expressions.

### Listing: Example of a user-defined absolute and relocatable SECTION

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec.

label2: EQU 5 ; label2 is assigned value 5.

label3: EQU label1 ; label3 is assigned the address of label1.
```

## 8.3.2 External symbols

A symbol may be made external using the [XDEF - External Symbol Definition](#) assembler directive. In another source file, an [XREF - External Symbol Reference](#) assembler directive must reference it. Since its address is unknown in the referencing file, it is considered to be relocatable. See the following listing for an example of using `XDEF` and `XREF`.

### Listing: Examples of external symbols

```
XREF extLabel ; symbol defined in an other module.
              ; extLabel is imported in the current module

XDEF label ; symbol is made external for other modules
           ; label is exported from the current module

constSec: SECTION
label: DC.W 1, extLabel
```

## 8.3.3 Undefined symbols

## Constants

If a label is neither defined in the source file nor declared external using `XREF`, the Assembler considers it to be undefined and generates an error message. The following listing shows an example of an undeclared label.

### Listing: Example of an undeclared label

```
codeSec: SECTION
entry:
    NOP
    BNE  entry
    NOP
    JMP  end
    JMP  label ; <- Undeclared user-defined symbol: label
end:RTS
    END
```

## 8.3.4 Reserved symbols

Reserved symbols cannot be used for user-defined symbols.

Register names are reserved identifiers.

For the HC08 processor the reserved identifiers are listed in the following listing:

### Listing: Reserved identifiers for an HC(S)08 derivative

```
A, CCR, H, X, SP
```

The keywords `LOW` and `HIGH` are also reserved identifiers. They are used to refer to the low byte and the high byte of a memory location.

## 8.4 Constants

The Assembler supports integer and ASCII string constants.

### 8.4.1 Integer constants

The Assembler supports four representations of integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9).

Example: 5, 512, 1024

- A hexadecimal constant is defined by a dollar character ( \$ ) followed by a sequence of hexadecimal digits (0-9, a-f, A-F).

Example: \$5, \$200, \$400

- An octal constant is defined by the commercial at character ( @ ) followed by a sequence of octal digits (0-7).

Example: @5, @1000, @2000

- A binary constant is defined by a percent character followed by a sequence of binary digits (0-1)

**Example :**

%101, %1000000000, %10000000000

The default base for integer constant is initially decimal, but it can be changed using the [BASE - Set number base](#) assembler directive. When the default base is not decimal, decimal values cannot be represented, because they do not have a prefix character.

## 8.4.2 String constants

A string constant is a series of printable characters enclosed in single ( `'` ) or double quote ( `"` ). Double quotes are only allowed within strings delimited by single quotes. Single quotes are only allowed within strings delimited by double quotes. See the following listing for a variety of string constants.

### Listing: String constants

```
'ABCD', "ABCD", 'A', "'B", "A'B", 'A"B'
```

## 8.4.3 Floating-Point constants

The Macro Assembler does not support floating-point constants.

## 8.5 Operators

Operators recognized by the Assembler in expressions are:

- Addition and subtraction operators (binary)
- Multiplication, division and modulo operators (binary)
- Sign operators (unary)
- Shift operators (binary)
- Bitwise operators (binary)
- Bitwise operators (unary)
- Logical operators (unary)
- Relational operators (binary)
- HIGH operator
- HIGH\_6\_13 Operator
- LOW operator
- MAP\_ADDR\_6 Operator
- PAGE operator
- Force operator (unary)
- Operator precedence

### 8.5.1 Addition and subtraction operators (binary)

The addition and subtraction operators are + and -, respectively.

#### Syntax

Addition:  
<operand> + <operand>

Subtraction:  
<operand> - <operand>

#### Description

The + operator adds two operands, whereas the - operator subtracts them. The operands can be any expression evaluating to an absolute or relocatable expression.

Addition between two relocatable operands is not allowed.

#### Example

See the following listing for an example of addition and subtraction operators.

### Listing: Addition and subtraction operators

```
$A3216 + $42 ; Addition of two absolute operands (= $A3258)
labelB - $10 ; Subtraction with value of `labelB'
```

## 8.5.2 Multiplication, division and modulo operators (binary)

The multiplication, division, and modulo operators are \*, /, and %, respectively.

### Syntax

```
    Multiplication:
<operand> * <operand>

    Division:
<operand> / <operand>

    Modulo:
<operand> % <operand>
```

### Description

The \* operator multiplies two operands, the / operator performs an integer division of the two operands and returns the quotient of the operation. The % operator performs an integer division of the two operands and returns the remainder of the operation

The operands can be any expression evaluating to an absolute expression. The second operand in a division or modulo operation cannot be zero.

### Example

See the following listing for an example of the multiplication, division, and modulo operators.

### Listing: Multiplication, division, and modulo operators

```
23 * 4      ; multiplication (= 92)
23 / 4      ; division (= 5)

23 % 4      ; remainder(= 3)
```

## 8.5.3 Sign operators (unary)

## Operators

The (unary) sign operators are + and -.

### Syntax

Plus:  
+<operand>

Minus:  
-<operand>

### Description

The + operator does not change the operand, whereas the - operator changes the operand to its two's complement. These operators are valid for absolute expression operands.

### Example

See the following listing for an example of the unary sign operators.

#### Listing: Unary sign operators

```
+$32      ; ( = $32)
-$32      ; ( = $CE = -$32)
```

## 8.5.4 Shift operators (binary)

The binary shift operators are << and >>.

### Syntax

Shift left:  
<operand> << <count>

Shift right:  
<operand> >> <count>

### Description

The << operator shifts its left operand left by the number of bits specified in the right operand.

The >> operator shifts its left operand right by the number of bits specified in the right operand.

The operands can be any expression evaluating to an absolute expression.

## Example

See the following listing for an example of the binary shift operators.

### Listing: Binary shift operators

```

$25 << 2    ; shift left (= $94)
$A5 >> 3    ; shift right(= $14)

```

## 8.5.5 Bitwise operators (binary)

The binary bitwise operators are `&`, `|`, and `^`.

### Syntax

```

Bitwise AND:
Bitwise OR:
<operand> | <operand>

Bitwise XOR:
<operand> ^ <operand>

```

### Description

The `&` operator performs an AND between the two operands on the bit level.

The `|` operator performs an OR between the two operands on the bit level.

The `^` operator performs an XOR between the two operands on the bit level.

The operands can be any expression evaluating to an absolute expression.

### Example

See the following listing for an example of the binary bitwise operators

### Listing: Binary bitwise operators

```

$E & 3      ; = $2 (%1110 & %0011 = %0010)
$E | 3      ; = $F (%1110 | %0011 = %1111)

$E ^ 3      ; = $D (%1110 ^ %0011 = %1101)

```

## 8.5.6 Bitwise operators (unary)

The unary bitwise operator is `~`.

## Syntax

One's complement:  
~<operand>

## Description

The ~ operator evaluates the one's complement of the operand.

The operand can be any expression evaluating to an absolute expression.

## Example

See the following listing for an example of the unary bitwise operator.

### Listing: Unary bitwise operator

```
~$C ; = $FFFFFFF3 (~%00000000 00000000 00000000 00001100
                    =%11111111 11111111 11111111 11110011)
```

## 8.5.7 Logical operators (unary)

The unary logical operator is !.

## Syntax

Logical NOT: !<operand>

## Description

The ! operator returns 1 (true) if the operand is 0, otherwise it returns 0 (false).

The operand can be any expression evaluating to an absolute expression.

## Example

See the following listing for an example of the unary logical operator.

### Listing: Unary logical operator

```
!(8<5) ; = $1 (TRUE)
```

## 8.5.8 Relational operators (binary)

The binary relational operators are =, ==, !=, <>, <, <=, >, and >=.

## Syntax



```

Equal:
<operand> = <operand>

<operand> == <operand>

Not equal:
<operand> != <operand>

                                <operand> <> <operand>

Less than:                        <operand> < <operand>

Less than or equal:
<operand> <= <operand>

Greater than:
<operand> > <operand>

Greater than or equal:
<operand> >= <operand>

```

## Description

These operators compare two operands and return 1 if the condition is true or 0 if the condition is false.

The operands can be any expression evaluating to an absolute expression.

## Example

See the following listing for an example of the binary relational operators

### Listing: Binary relational operators

```

3 >= 4      ; = 0 (FALSE)
label = 4   ; = 1 (TRUE) if label is 4, 0 or (FALSE) otherwise.
9 < $B      ; = 1 (TRUE)

```

## 8.5.9 HIGH operator

The HIGH operator is HIGH.

### Syntax

```
High Byte: HIGH(<operand>)
```

### Description

This operator returns the high byte of the address of a memory location.

### Example

## Operators

Assume `data1` is a word located at address `$1050` in the memory.

```
LDA #HIGH(data1)
```

This instruction will load the immediate value of the high byte of the address of `data1` (`$10`) in register A.

```
LDA HIGH(data1)
```

This instruction will load the direct value at memory location of the higher byte of the address of `data1` (i.e., the value in memory location `$10`) in register A.

### 8.5.10 HIGH\_6\_13 Operator

#### Syntax

```
High Byte: HIGH_6_13(<operand>)
```

#### Description

This operator returns the high byte of a 14-bit address of a memory location.

#### Example

Assume `data1` is a word located at address `$1010` in the memory.

```
LDA #HIGH_6_13(data1)
```

This instruction will load the value `$40` in the accumulator.

### 8.5.11 LOW operator

The LOW operator is `LOW`.

#### Syntax

```
LOW Byte: LOW(<operand>)
```

#### Description

This operator returns the low byte of the address of a memory location.

#### Example

Assume `data1` is a word located at address `$1050` in the memory.

```
LDA #LOW(data1)
```

This instruction will load the immediate value of the lower byte of the address of `data1` (`$50`) in register A.

```
LDA LOW(data1)
```

This instruction will load the direct value at memory location of the lower byte of the address of `data1` (i.e., the value in memory location `$50`) in register A.

## 8.5.12 MAP\_ADDR\_6 Operator

### Syntax

```
MAP_ADDR_6(<operand>)
```

### Description

This operator returns the lower 6 bits for a memory location. It should be used to determine the offset in the paging window for a certain memory address. Note that the operator automatically adds the offset of the base of the paging window (`$C0`).

### Example

```
MOV    #HIGH_6_13(data), $001F
```

```
STA    MAP_ADDR_6(data)
```

In this example, the RS08 PAGE register (mapped at `$001F`) is loaded with the memory page corresponding to `data` and then the value contained in the accumulator is stored at the address pointed by `data`.

## 8.5.13 PAGE operator

The PAGE operator is `PAGE`.

### Syntax

```
PAGE Byte: PAGE(<operand>)
```

### Description

This operator returns the page byte of the address of a memory location.

### Example

## Operators

Assume `data1` is a word located at address `$28050` in the memory.

```
LDA #PAGE(data1)
```

This instruction will load the immediate value of the page byte of the address of `data1` (`$2`).

```
LDA PAGE(data1)
```

This instruction will load the direct value at memory location of the page byte of the address of `data1` (i.e., the value in memory location `$2`).

### NOTE

The `PAGE` keyword does not refer to the RS08 `PAGE` register but to the `PAGE` operator described above.

## 8.5.14 Force operator (unary)

### Syntax

8-bit address:  
`<<operand>` or  
`<operand>.B`

16-bit address:  
`><operand>` or  
`<operand>.W`

### Description

The `<` or `.B` operators force direct addressing mode, whereas the `>` or `.W` operators force extended addressing mode.

Use the `<` operator to force 8-bit indexed or 8-bit direct addressing mode for an instruction.

Use the `>` operator to force 16-bit indexed or 16-bit extended addressing mode for an instruction.

The operand can be any expression evaluating to an absolute or relocatable expression.

### Example

```

<label      ; label is an 8-bit address.

label.B     ; label is an 8-bit address.

>label      ; label is an 16-bit address.

label.W     ; label is an 16-bit address.

```

For the RS08 the < operand forces the operand to short or tiny addressing mode (depending on the instruction in which it is used). The same result can be obtained by adding .S or .T to the referred symbol. The > operator forces an address to 8 bits, even if it fits in 4 or 5 bits (so short or tiny addressing modes can be used).

### 8.5.15 Operator precedence

The following table lists the operator precedence rules for ANSI - C operators.

**Table 8-6. Operator precedence priorities**

Operator	Description	Associativity
()	Parenthesis	Right to Left
~ + -	One's complement Unary Plus Unary minus	Left to Right
* / %	Integer multiplication Integer division Integer modulo	Left to Right
+ -	Integer addition Integer subtraction	Left to Right
<< >>	Shift Left Shift Right	Left to Right
< <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to Right
=, == !=, <>	Equal to Not Equal to	Left to Right
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
	Bitwise OR	Left to Right

## 8.6 Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols

## expression

- External symbols
- The special symbol `\*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the `DC` directive:

```
DC.W 1, 2, *-2
```

Once a valid expression has been fully evaluated by the Assembler, it is reduced as one of the following type of expressions:

- **Absolute expression** : The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant.
- **Simple relocatable expression** : The expression evaluates to an absolute offset from the start of a single relocatable section.
- **Complex relocatable expression**: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The Assembler does not support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in `XREF` directives, which are assumed to be relocatable symbols.

### 8.6.1 Absolute expression

An absolute expression is an expression involving constants or known absolute labels or expressions. An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

See the following listing for an example of an absolute expression.

#### Listing: Absolute expression

```
Base: SET $100
Label: EQU Base * $5 + 3
```

Expressions involving the difference between two relocatable symbols defined in the same file and in the same section evaluate to an absolute expression. An expression as `label2-label1` can be translated as:

#### Listing: Interpretation of label2-label1: difference between two relocatable symbols

```
(<offset label2> + <start section address >) -
(<offset label1> + <start section address >)
```

This can be simplified to the following listing:

## Listing: Simplified result for the difference between two relocatable symbols

```
<offset label2> + <start section address > -
<offset label1> - <start section address>

= <offset label2> - <offset label1>
```

### 8.6.1.1 Example

In the example in the following listing, the expression `tabEnd-tabBegin` evaluates to an absolute expression and is assigned the value of the difference between the offset of `tabEnd` and `tabBegin` in the section `DataSec`.

## Listing: Absolute expression relating the difference between two relocatable symbols

```
DataSec: SECTION
tabBegin: DS.B 5

tabEnd: DS.B 1

ConstSec: SECTION

label: EQU tabEnd-tabBegin ; Absolute expression

CodeSec: SECTION

entry: NOP
```

### 8.6.2 Simple relocatable expression

A simple relocatable expression results from an operation such as one of the following:

- `<relocatable expression> + <absolute expression>`
- `<relocatable expression> - <absolute expression>`
- `<absolute expression> + <relocatable expression>`

## Listing: Example of relocatable expression

```
                XREF XtrnLabel
DataSec: SECTION

tabBegin: DS.B 5

tabEnd: DS.B 1

CodeSec: SECTION

entry:

                LDA tabBegin+2 ; Simple relocatable expression
```

```

BRA *-3           ; Simple relocatable expression

LDA XtrnLabel+6   ; Simple relocatable expression
    
```

### 8.6.3 Unary operation result

The following table describes the type of an expression according to the operator in an unary operation:

**Table 8-7. Expression type resulting from operator and operand type**

Operator	Operand	Expression
-, !, ~	absolute	absolute
-, !, ~	relocatable	complex
+	absolute	absolute
+	relocatable	relocatable

### 8.6.4 Binary operations result

The following table describes the type of an expression according to the left and right operators in a binary operation:

**Table 8-8. Expression type resulting from operator and their operands**

Operator	Left Operand	Right Operand	Expression
-	absolute	absolute	absolute
-	relocatable	absolute	relocatable
-	absolute	relocatable	complex
-	relocatable	relocatable	absolute
+	absolute	absolute	absolute
+	relocatable	absolute	relocatable
+	absolute	relocatable	relocatable
+	relocatable	relocatable	complex
*, /, %, <<, >>,  , &, ^	absolute	absolute	absolute
*, /, %, <<, >>,  , &, ^	relocatable	absolute	complex
*, /, %, <<, >>,  , &, ^	absolute	relocatable	complex
*, /, %, <<, >>,  , &, ^	relocatable	relocatable	complex



## 8.7 Translation limits

The following limitations apply to the Macro Assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Include may be nested up to 50.
- The maximum line length is 1023.



# Chapter 9

## Assembler Directives

There are different classes of assembler directives. The following tables give you an overview over the different directives and their classes:

### 9.1 Directive Overview

This section provides an overview of assembler directives.

#### 9.1.1 Section-Definition directives

The following table lists the directives to define new sections.

**Table 9-1. Directives for defining sections**

Directive	Description
<a href="#">ORG - Set Location Counter</a>	Define an absolute section
<a href="#">SECTION - Declare Relocatable Section</a>	Define a relocatable section
<a href="#">OFFSET - Create absolute symbols</a>	Define an offset section

#### 9.1.2 Constant-Definition directives

The following table lists the directives to define assembly constants.

**Table 9-2. Directives for defining constants**

Directive	Description
<a href="#">EQU - Equate symbol value</a>	Assign a name to an expression (cannot be redefined)
<a href="#">SET - Set Symbol Value</a>	Assign a name to an expression (can be redefined)

### 9.1.3 Data-Allocation directives

The following table lists the directives to allocate variables.

**Table 9-3. Directives for allocating variables**

Directive	Description
<a href="#">DC - Define Constant</a>	Define a constant variable
<a href="#">DCB - Define Constant Block</a>	Define a constant block
<a href="#">DS - Define Space</a>	Define storage for a variable
<a href="#">RAD50 - RAD50-encoded string constants</a>	RAD50 encoded string constants

### 9.1.4 Symbol-Linkage directives

The following table lists the symbol-linkage directives to export or import global symbols.

**Table 9-4. Symbol linkage directives**

Directive	Description
<a href="#">ABSENTRY - Application entry point</a>	Specify the application entry point when an absolute file is generated
<a href="#">XDEF - External Symbol Definition</a>	Make a symbol public (visible from outside)
<a href="#">XREF - External Symbol Reference</a>	Import reference to an external symbol.
<a href="#">XREFB - External Reference for Symbols located on the Direct Page</a>	Import reference to an external symbol located on the direct page.

## 9.1.5 Assembly-Control directives

The following table lists the assembly-control general purpose directives to control the assembly process.

**Table 9-5. Assembly control directives**

Directive	Description
<a href="#">ALIGN - Align Location Counter</a>	Define Alignment Constraint
<a href="#">BASE - Set number base</a>	Specify default base for constant definition
<a href="#">END - End assembly</a>	End of assembly unit
<a href="#">ENDFOR - End of FOR block</a>	End of FOR block
<a href="#">EVEN - Force word alignment</a>	Define 2-byte alignment constraint
<a href="#">FAIL - Generate Error message</a>	Generate user defined error or warning messages
<a href="#">FOR - Repeat assembly block</a>	Repeat assembly blocks
<a href="#">INCLUDE - Include text from another file</a>	Include text from another file.
<a href="#">LONGEVEN - Forcing Long-Word alignment</a>	Define 4 Byte alignment constraint

## 9.1.6 Listing-File Control directives

The following table lists the listing-file control directives to control the generation of the assembler listing file.

**Table 9-6. Listing-file control directives**

Directive	Description
<a href="#">CLIST - List conditional assembly</a>	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
<a href="#">LIST - Enable Listing</a>	Specify that all subsequent instructions must be inserted in the listing file.
<a href="#">LLEN - Set Line Length</a>	Define line length in assembly listing file.
<a href="#">MLIST - List macro expansions</a>	Specify if the macro expansions must be inserted in the listing file.
<a href="#">NOLIST - Disable Listing</a>	Specify that all subsequent instruction must not be inserted in the listing file.
<a href="#">NOPAGE - Disable Paging</a>	Disable paging in the assembly listing file.
<a href="#">PAGE - Insert Page break</a>	Insert page break.
<a href="#">PLEN - Set Page Length</a>	Define page length in the assembler listing file.
<a href="#">SPC - Insert Blank Lines</a>	Insert an empty line in the assembly listing file.
<a href="#">TABS - Set Tab Length</a>	Define number of character to insert in the assembler listing file for a TAB character.

*Table continues on the next page...*

**Table 9-6. Listing-file control directives (continued)**

Directive	Description
<a href="#">TITLE - Provide Listing Title</a>	Define the user defined title for the assembler listing file.

## 9.1.7 Macro Control directives

The following table lists the macro control directives, used for the definition and expansion of macros.

**Table 9-7. Macro control directives**

Directive	Description
<a href="#">ENDM - End macro definition</a>	End of user defined macro.
<a href="#">MACRO - Begin macro definition</a>	Start of user defined macro.
<a href="#">MEXIT - Terminate Macro Expansion</a>	Exit from macro expansion.

## 9.1.8 Conditional Assembly directives

The following table lists the conditional assembly directives, used for conditional assembling.

**Table 9-8. Conditional assembly directives**

Directive	Description
<a href="#">ELSE - Conditional assembly</a>	alternate block
<a href="#">ENDIF - End conditional assembly</a>	End of conditional block
<a href="#">IF - Conditional assembly</a>	Start of conditional block. A boolean expression follows this directive.
<a href="#">IFcc - Conditional assembly</a>	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater than or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less than or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined

*Table continues on the next page...*

**Table 9-8. Conditional assembly directives (continued)**

Directive	Description
IFNE	Test if an expression is not null.

## 9.2 Detailed descriptions of all assembler directives

The remainder of the chapter covers the detailed description of all available assembler directives.

### 9.2.1 ABSENTRY - Application entry point

#### Syntax

```
ABSENTRY <label>
```

#### Synonym

None

#### Description

This directive is used to specify the application Entry Point when the Assembler directly generates an absolute file. The `-FA2` assembly option - ELF/DWARF 2.0 Absolute File - must be enabled.

Using this directive, the entry point of the assembly application is written in the header of the generated absolute file. When this file is loaded in the debugger, the line where the entry point label is defined is highlighted in the source window.

This directive is ignored when the Assembler generates an object file.

#### NOTE

This instruction only affects the loading on an application by a debugger. It tells the debugger which initial PC should be used. In order to start the application on a target, initialize the Reset vector.

If the example in the following listing is assembled using the `-FA2` assembler option, an ELF/DWARF 2.0 Absolute file is generated.

**Listing: Using ABSENTRY to specify an application entry point**

```

        ABSENTRY entry
        ORG   $fffe

Reset:  DC.W  entry

        ORG   $70

entry:  NOP

        NOP

main:   RSP

        NOP

        BRA  main
    
```

According to the `ABSENTRY` directive, the entry point will be set to the address of `entry` in the header of the absolute file.

## 9.2.2 ALIGN - Align Location Counter

### Syntax

```
ALIGN <n>
```

### Synonym

None

### Description

This directive forces the next instruction to a boundary that is a multiple of `<n>`, relative to the start of the section. The value of `<n>` must be a positive number between 1 and 32767. The `ALIGN` directive can force alignment to any size. The filling bytes inserted for alignment purpose are initialized with `^ \0 ^`.

`ALIGN` can be used in code or data sections.

### Example

The example shown in the following listing, aligns the `HEX` label to a location, which is a multiple of 16 (in this case, location `00010` (Hex))



## Listing: Aligning the HEX Label to a Location

Assembler Abs. Rel.	Loc	Obj. code	Source line
1	1		
2	2	000000 6869 6768	DC.B "high"
3	3	000004 0000 0000	ALIGN 16
		000008 0000 0000	
		00000C 0000 0000	
4	4		
5	5		
6	6	000010 7F	HEX: DC.B 127 ; HEX is allocated
7	7		; on an address,
8	8		; which is a
9	9		; multiple of 16.

### 9.2.3 BASE - Set number base

#### Syntax

```
BASE <n>
```

#### Synonym

None

#### Description

The directive sets the default number base for constants to <n>. The operand <n> may be prefixed to indicate its number base; otherwise, the operand is considered to be in the current default base. Valid values of <n> are 2, 8, 10, 16. Unless a default base is specified using the `BASE` directive, the default number base is decimal.

#### Example

See the following listing, for examples of setting the number base.

#### Listing: Setting the number base

## Detailed descriptions of all assembler directives

4	4		base	10	; default base: decimal
5	5	000000 64	dc.b	100	
6	6		base	16	; default base: hex
7	7	000001 0A	dc.b	0a	
8	8		base	2	; default base: binary
9	9	000002 04	dc.b	100	
10	10	000003 04	dc.b	%100	
11	11		base	@12	; default base: decimal
12	12	000004 64	dc.b	100	
13	13		base	\$a	; default base: decimal
14	14	000005 64	dc.b	100	
15	15				
16	16		base	8	; default base: octal
17	17	000006 40	dc.b	100	

Be careful. Even if the base value is set to 16, hexadecimal constants terminated by a `D` must be prefixed by the `$` character, otherwise they are supposed to be decimal constants in old style format. For example, constant `45D` is interpreted as decimal constant 45, not as hexadecimal constant `45D`.

## 9.2.4 CLIST - List conditional assembly

### Syntax

```
CLIST [ON|OFF]
```

### Synonym

None

### Description

The `CLIST` directive controls the listing of subsequent conditional assembly blocks. It precedes the first directive of the conditional assembly block to which it applies, and remains effective until the next `CLIST` directive is read.

When the `ON` keyword is specified in a `CLIST` directive, the listing file includes all directives and instructions in the conditional assembly block, even those which do not generate code (which are skipped).

When the `OFF` keyword is entered, only the directives and instructions that generate code are listed.

As soon as the `-L: Generate a listing file` assembler option is activated, the Assembler defaults to `CLIST ON`.

### Example

The following listing is an example where the `CLIST OFF` option is used.

#### Listing: Listing file with CLIST OFF

```

CLIST OFF
Try: EQU    0

    IFEQ    Try

        LDA    #103

    ELSE

        LDA    #0

    ENDIF

```

The following listing is the corresponding listing file.

#### Listing: Example assembler listing where CLIST ON is used

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
2	2		0000 0000	Try: EQU    0
3	3		0000 0000	IFEQ    Try
4	4	000000	A667	LDA    #103
5	5			ELSE
7	7			ENDIF

The following listing is a listing file using `CLIST ON`.

#### Listing: CLIST ON is selected

```

CLIST ON
Try: EQU    0

    IFEQ    Try

        LDA    #103

    ELSE

        LDA    #0

    ENDIF

```

The following listing is the corresponding listing file.

**Listing: Example assembler listing where CLIST ON is used**

Abs.	Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----	-----
2	2		0000 0000	Try: EQU 0
3	3		0000 0000	IFEQ Try
4	4	000000	A667	LDA #103
5	5			ELSE
6	6			LDA #0
7	7			ENDIF
8	8			

## 9.2.5 DC - Define Constant

### Syntax

```
[<label>:] DC [<size>] <expression> [, <expression>]...
```

where <size> = B (default), W, or L.

### Synonym

```
DCW (= 2 byte DCs), DCL (= 4 byte DCs),
FCB (= DC.B), FDB (= 2 byte DCs),
FQB (= 4 byte DCs)
```

### Description

The DC directive defines constants in memory. It can have one or more <expression> operands, which are separated by commas. The <expression> can contain an actual value (binary, octal, decimal, hexadecimal, or ASCII). Alternatively, the <expression> can be a symbol or expression that can be evaluated by the Assembler as an absolute or simple relocatable expression. One memory block is allocated and initialized for each expression.

The following rules apply to size specifications for DC directives:

- **DC.B:** One byte is allocated for numeric expressions. One byte is allocated per ASCII character for strings.

**Listing: Example for DC.B**

```
000000 4142 4344   Label: DC.B "ABCDE"
000004 45

000005 0A0A 010A           DC.B %1010, @12, 1, $A
```

- **DC.W:** Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary.

**Listing: Example for DC.W**

```
000000 0041 4243   Label: DC.W "ABCDE"
000004 4445

000006 000A 000A           DC.W %1010, @12, 1, $A

00000A 0001 000A

00000E xxxx           DC.W Label
```

- **DC.L:** Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary.

**Listing: Example for DC.L**

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445

000008 0000 000A           DC.L %1010, @12, 1, $A

00000C 0000 000A

000010 0000 0001

000014 0000 000A

000018 xxxx xxxx           DC.L Label
```

If the value in an operand expression exceeds the size of the operand, the assembler truncates the value and generates a warning message.

**See also**

Assembler directives:

- [DCB - Define Constant Block](#)
- [DS - Define Space](#)
- [ORG - Set Location Counter](#)
- [SECTION - Declare Relocatable Section](#)

## 9.2.6 DCB - Define Constant Block

### Syntax

```
[<label>:] DCB [.<size>] <count>, <value>
```

where

```
<size> =  
B (default),  
W, or  
L.
```

### Description

The `DCB` directive causes the Assembler to allocate a memory block initialized with the specified `<value>`. The length of the block is `<size> * <count>`.

`<count>` may not contain undefined, forward, or external references. It may range from 1 to 4096.

The value of each storage unit allocated is the sign-extended expression `<value>`, which may contain forward references. The `<count>` cannot be relocatable. This directive does not perform any alignment.

The following rules apply to size specifications for `DCB` directives:

- `DCB.B`: One byte is allocated for numeric expressions.
- `DCB.W`: Two bytes are allocated for numeric expressions.
- `DCB.L`: Four bytes are allocated for numeric expressions.

#### Listing: Examples of DCB directives

```
000000 FFFF FF      Label: DCB.B 3, $FF  
000003 FFFE FFFE      DCB.W 3, $FFFE  
  
000007 FFFE  
  
000009 0000 FFFE      DCB.L 3, $FFFE  
  
00000D 0000 FFFE  
  
000011 0000 FFFE
```

### See also

#### Assembler directives :

- [DC - Define Constant](#)

- DS - Define Space
- ORG - Set Location Counter
- SECTION - Declare Relocatable Section

## 9.2.7 DS - Define Space

### Syntax

```
[<label>:] DS[.<size>] <count>
```

where <size> = B (default), W, or L.

### Synonym

```
RMB (= DS.B)
```

```
RMD (2 bytes)
```

```
RMQ (4 bytes)
```

### Description

The DS directive is used to reserve memory for variables, as listed in the following listing. The content of the memory reserved is not initialized. The length of the block is <size> \* <count>.

<count> may not contain undefined, forward, or external references. It may range from 1 to 4096.

### Listing: Examples of DS directives

```
Counter: DS.B 2 ; 2 continuous bytes in memory
         DS.B 2 ; 2 continuous bytes in memory
         ; can only be accessed through the label Counter
         DS.W 5 ; 5 continuous words in memory
```

The label Counter references the lowest address of the defined storage area.

### NOTE

Storage allocated with a DS directive may end up in constant data section or even in a code section, if the same section contains constants or code as well. The Assembler allocates only a complete section at once.

## Example

In the following listing, a variable, a constant, and code were put in the same section. Because code has to be in ROM, then all three elements must be put into ROM.

### Listing: Poor memory allocation

```
; How it should NOT be done ...
Counter:      DS 1      ; 1-byte used

InitialCounter: DC.B $f5 ; constant $f5

main:         NOP      ; NOP instruction
```

In order to allocate them separately, put them in different sections, as listed in the following listing:

### Listing: Proper memory allocation

```
DataSect:     SECTION   ; separate section for variables
Counter:      DS 1      ; 1-byte used

ConstSect:    SECTION   ; separate section for constants

InitialCounter: DC.B $f5 ; constant $f5

CodeSect:     SECTION   ; section for code

main:         NOP      ; NOP instruction
```

An `ORG` directive also starts a new section.

### See also

- [DC - Define Constant](#)
- [ORG - Set Location Counter](#)
- [SECTION - Declare Relocatable Section](#)

## 9.2.8 ELSE - Conditional assembly

### Syntax

```
IF <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

### Synonym

ELSEC



## Description

If `<condition>` is true, the statements between `IF` and the corresponding `ELSE` directive are assembled (generate code).

If `<condition>` is false, the statements between `ELSE` and the corresponding `ENDIF` directive are assembled. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

## Example

The following listing is an example of the use of conditional assembly directives:

### Listing: Various conditional assembly directives

```
Try: EQU    1
     IF Try  != 0

         LDA  #103

     ELSE

         LDA  #0

     ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `lda #103` instruction is assembled. Changing the operand of the `EQU` directive to `0` causes the `lda #0` instruction to be assembled instead.

### Listing: Output listing

Abs.	Rel.	Loc	Obj. code	Source line
1	1		0000 0001	Try: EQU 1
2	2		0000 0001	IF Try != 0
3	3	000000	A667	LDA #103
4	4			ELSE
6	6			ENDIF

## 9.2.9 END - End assembly

### Syntax

END

## Synonym

None

## Description

The `END` directive indicates the end of the source code. Subsequent source statements in this file are ignored. The `END` directive in included files skips only subsequent source statements in this include file. The assembly continues in the including file in a regular way.

## Example

The `END` statement in the following listing causes any source code after the `END` statement to be ignored, as in the next listing.

### Listing: Source File

```
Label: DC.W $1234
      DC.W $5678

      END

      DC.W $90AB ; no code generated
      DC.W $CDEF ; no code generated
```

### Listing: Generated listing file

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1	000000	1234	Label: DC.W \$1234
2	2	000002	5678	DC.W \$5678

## 9.2.10 ENDFOR - End of FOR block

### Syntax

ENDFOR

### Synonym

None

## Description

The `ENDFOR` directive indicates the end of a `FOR` block.

### NOTE

The `FOR` directive is only available when the `-Compat=b` assembler option is used. Otherwise, the `FOR` directive is not supported.

## Example

See [Listing: Using the FOR directive in a loop](#) in the `FOR` section.

## See also

Assembler directives:

- [FOR - Repeat assembly block](#)
- [-Compat: Compatibility modes](#)

## 9.2.11 ENDIF - End conditional assembly

### Syntax

```
ENDIF
```

### Synonym

```
ENDC
```

## Description

The `ENDIF` directive indicates the end of a conditional block. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

## Example

See [Listing: IF and ENDIF](#) in the `IF` section.

## See also

[IF - Conditional assembly](#) assembler directive

## 9.2.12 ENDM - End macro definition

### Syntax

```
ENDM
```

### Synonym

None

### Description

The `ENDM` directive terminates the macro definition.

### Example

The `ENDM` statement in the following listing terminates the `cpChar` macro.

#### Listing: Using ENDM to terminate a macro definition

```
cpChar:  MACRO
         LDA   \1
         STA   \2
         ENDM

CodeSec: SECTION

Start:

        cpChar char1, char2
        LDA   char1
        STA   char2
```

## 9.2.13 EQU - Equate symbol value

### Syntax

```
<label>: EQU <expression>
```

### Synonym

None

## Description

The `EQU` directive assigns the value of the `<expression>` in the operand field to `<label>`. The `<label>` and `<expression>` fields are both required, and the `<label>` cannot be defined anywhere else in the program. The `<expression>` cannot include a symbol that is undefined or not yet defined.

The `EQU` directive does not allow forward references.

## Example

See the following listing for examples of using the `EQU` directive.

### Listing: Using EQU to set variables

```
0000 0014  MaxElement: EQU  20
0000 0050  MaxSize:    EQU  MaxElement * 4

                Time:   DS.B  3

0000 0000  Hour:    EQU   Time    ; first byte addr.
0000 0002  Minute: EQU   Time+1  ; second byte addr
0000 0004  Second: EQU   Time+2  ; third byte addr
```

## 9.2.14 EVEN - Force word alignment

### Syntax

```
EVEN
```

### Synonym

None

### Description

This directive forces the next instruction to the next even address relative to the start of the section. `EVEN` is an abbreviation for `ALIGN 2`. Some processors require word and long word operations to begin at even address boundaries. In such cases, the use of the `EVEN` directive ensures correct alignment. Omission of this directive can result in an error message.

### Example

See the following listing for instances where the EVEN directive causes padding bytes to be inserted.

### Listing: Using the Force Word Alignment Directive

Abs.	Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----	-----
1	1	000000		ds.b 4
2	2			; location count has an even value
3	3			; no padding byte inserted.
4	4			even
5	5	000004		ds.b 1
6	6			; location count has an odd value
7	7			; one padding byte inserted.
8	8	000005		even
9	9	000006		ds.b 3
10	10			; location count has an odd value
11	11			; one padding byte inserted.
12	12	000009		even
13	13		0000 000A	aaa: equ 10

### See also

[ALIGN - Align Location Counter](#) assembly directive

## 9.2.15 FAIL - Generate Error message

### Syntax

```
FAIL <arg>|<string>
```

### Synonym

None

### Description

There are three modes of the FAIL directive, depending upon the operand that is specified:

- If <arg> is a number in the range [0-499], the Assembler generates an error message, including the line number and argument of the directive. The Assembler does not generate an object file.
- If <arg> is a number in the range [500-0xFFFFFFFF], the Assembler generates a warning message, including the line number and argument of the directive.
- If a string is supplied as an operand, the Assembler generates an error message, including the line number and the <string>. The Assembler does not generate an object file.
- The `FAIL` directive is primarily intended for use with conditional assembly to detect user-defined errors or warning conditions.

## Examples

The assembly code in the following listing generates the error messages in the next listing. The value of the operand associated with the `FAIL 200` or `FAIL 600` directives determines (1) the format of any warning or error message and (2) whether the source code segment will be assembled.

### Listing: Example source code

```

cpChar: MACRO
    IFC "\1", ""

        FAIL 200

        MEXIT

    ELSE

        LDA  \1

    ENDIF

    IFC "\2", ""

        FAIL 600

    ELSE

        STA  \2

    ENDIF

ENDM

codSec: SECTION

Start:

    cpChar char1

```

### Listing: Error messages resulting from assembling the source code

```

>> in "C:\Freescale\demo\warnfail.asm", line 13, col 19, pos 226
    IFC "\2", ""

```

## Detailed descriptions of all assembler directives

```
FAIL 600
```

```
^
```

```
WARNING A2332: FAIL found
```

```
Macro Call :          FAIL 600
```

The following listing is another assembly code example which again incorporates the `FAIL 200` and the `FAIL 600` directives.

### Listing: Example source code

```
cpChar: MACRO
    IFC "\1", ""

        FAIL 200

        MEXIT

    ELSE

        LDA \1

    ENDIF

    IFC "\2", ""

        FAIL 600

    ELSE

        STA \2

    ENDIF

ENDM

codeSec: SECTION

Start:

    cpChar, char2
```

The following listing is the error message that was generated as a result of assembling the source code in the above listing.

### Listing: Error messages resulting from assembling the source code

```
>> in "C:\Freescale\demo\errfail.asm", line 6, col 19, pos 96
    IFC "\1", ""

        FAIL 200

        ^

ERROR A2329: FAIL found

Macro Call :          FAIL 200
```

The following listing has additional uses of the `FAIL` directive. In this example, the `FAIL string` and `FAIL 600` directives are used.



## Listing: Example source code

```
cpChar: MACRO
    IFC "\1", ""

        FAIL "A character must be specified as first parameter"

        MEXIT

    ELSE

        LDA \1

    ENDIF

    IFC "\2", ""

        FAIL 600

    ELSE

        STA \2

    ENDIF

ENDM

codeSec: SECTION

Start:

    cpChar, char2
```

The following listing shows the error messages generated from the assembly code as a result of the `FAIL` directive.

## Listing: Error messages resulting from assembling the source code

```
>> in "C:\Freescale\demo\failmes.asm", line 7, col 17, pos 110
    IFC "\1", ""

        FAIL "A character must be specified as first parameter"
            ^

ERROR A2338: A character must be specified as first parameter
Macro Call :   FAIL "A character must be specified as first parameter"
```

## 9.2.16 FOR - Repeat assembly block

### Syntax

```
FOR <label>=<num> TO <num>
```

ENDFOR

## Synonym

None

## Description

The `FOR` directive is an inline macro because it can generate multiple lines of assembly code from only one line of input code.

`FOR` takes an absolute expression and assembles the portion of code following it, the number of times represented by the expression. The `FOR` expression may be either a constant or a label previously defined using `EQU` or `SET`.

### NOTE

The `FOR` directive is only available when the `-Compat=b` assembly option is used. Otherwise, the `FOR` directive is not supported.

## Example

The following listing is an example of using `FOR` to create a 5-repetition loop.

### Listing: Using the FOR directive in a loop

```
FOR label=2 TO 6
  DC.B label*7

ENDFOR
```

### Listing: Resulting output listing

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			FOR label=2 TO 6
2	2			DC.B label*7
3	3			ENDFOR
4	2	000000	0E	DC.B label*7
5	3			ENDFOR
6	2	000001	15	DC.B label*7
7	3			ENDFOR
8	2	000002	1C	DC.B label*7
9	3			ENDFOR
10	2	000003	23	DC.B label*7

```
11      3          ENDFOR
12      2      000004 2A          DC.B  label*7
13      3          ENDFOR
```

## See also

[ENDFOR - End of FOR block](#)

[-Compat: Compatibility modes](#) assembler option

## 9.2.17 IF - Conditional assembly

### Syntax

```
IF <condition>

    [<assembly language statements>]

[ELSE]

    [<assembly language statements>]

ENDIF
```

### Synonym

None

### Description

If `<condition>` is true, the statements immediately following the `IF` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached. Then all the statements until the corresponding `ENDIF` directive are ignored. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

The expected syntax for `<condition>` is:

<condition> := <expression> <relation> <expression>

<relation> := =|!=|>|=|>|<=|<|<>

The <expression> must be absolute (It must be known at assembly time).

### Example

The following listing is an example of the use of conditional assembly directives

#### Listing: IF and ENDIF

```
Try: EQU    0
      IF Try != 0
          LDA    #103
      ELSE
          LDA    #0
      ENDIF
```

The value of Try determines the instruction to be assembled in the program. As shown, the `lda #0` instruction is assembled. Changing the operand of the `EQU` directive to one causes the `lda #103` instruction to be assembled instead. The following shows the listing provided by the Assembler for these lines of code:

#### Listing: Output listing after conditional assembly

```
1  1          0000 0000  Try: EQU    0
2  2          0000 0000  IF Try != 0

4  4                                ELSE
5  5  000000 A600          LDA    #0
6  6                                ENDIF
```

## 9.2.18 IFcc - Conditional assembly

### Syntax

IFcc <condition>

[<assembly language statements>]

```
[ELSE]
    [<assembly language statements>]
ENDIF
```

## Synonym

None

## Description

These directives can be replaced by the `IF` directive. If `ifcc <condition>` is true, the statements immediately following the `ifcc` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached, after which assembly moves to the statements following the `ENDIF` directive. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

The following table lists the available conditional types:

**Table 9-9. Conditional assembly types**

ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

## Example

The following listing is an example of the use of conditional assembler directives:

### Listing: Using the IFNE conditional assembler directive

```
Try: EQU    0
     IFNE   Try
         LDA    #103
     ELSE
         LDA    #0
     ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `lda #0` instruction is assembled. Changing the directive to `IFEQ` causes the `lda #103` instruction to be assembled instead.

The following listing shows the listing provided by the Assembler for these lines of code

**Listing: Output**

```

1  1          0000 0000   Try: EQU   0
2  2          0000 0000   IFNE  Try

4  4                               ELSE

5  5  000000 A600          LDA   #0

6  6                               ENDIF

```

## 9.2.19 INCLUDE - Include text from another file

### Syntax

```
INCLUDE <file specification>
```

### Synonym

None

### Description

This directive causes the included file to be inserted in the source input stream. The `<file specification>` is not case-sensitive and must be enclosed in quotation marks.

The Assembler attempts to open `<file specification>` relative to the current working directory. If the file is not found there, then it is searched for relative to each path specified in the [GENPATH: Search path for input file](#) environment variable.

### Example

```
INCLUDE "..\LIBRARY\macros.inc"
```

## 9.2.20 LIST - Enable Listing

## Syntax

LIST

## Synonym

None

## Description

Specifies that instructions following this directive must be inserted into the listing and into the debug file. This is a default option. The listing file is only generated if the [-L: Generate a listing file](#) assembler option is specified on the command line.

The source text following the LIST directive is listed until a [NOLIST - Disable Listing](#) or an [END - End assembly](#) assembler directive is reached.

This directive is not written to the listing and debug files.

## Example

The assembly source code using the LIST and NOLIST directives in the following listing generates the output listed in the next listing.

### Listing: Using the LIST and NOLIST assembler directives

```

aaa:   NOP
       LIST

bbb:   NOP

       NOP

       NOLIST

ccc:   NOP

       NOP

       LIST

ddd:   NOP           NOP
    
```

### Listing: Output

Abs.	Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----	-----
1	1	000000	9D	aaa: NOP
2	2			
4	4	000001	9D	bbb: NOP
5	5	000002	9D	NOP
6	6			

```

12  12  000005 9D          ddd:  NOP
13  13  000006 9D          NOP

```

## 9.2.21 LLEN - Set Line Length

### Syntax

```
LLEN<n>
```

### Synonym

None

### Description

Sets the number of characters from the source line that are included on the listing line to <n>. The values allowed for <n> are in the range [0 - 132]. If a value smaller than 0 is specified, the line length is set to 0. If a value bigger than 132 is specified, the line length is set to 132.

Lines of the source file that exceed the specified number of characters are truncated in the listing file.

### Example

The following listing shows the portion of code which generates the listing file in next listing. Notice that the `LLEN 24` directive causes the output at the location-counter line 7 to be truncated.

#### Listing: Example assembly source code using LLEN

```

DC.B  $55
LLEN  32

DC.W  $1234, $4567

LLEN  24

DC.W  $1234, $4567

EVEN

```

#### Listing: Formatted assembly output listing as a result of using LLEN

```

Abs. Rel.  Loc   Obj. code  Source line
-----

```



```

1   1   000000 55                DC.B  $55
2   2
4   4   000001 1234 4567        DC.W  $1234, $4567
5   5
7   7   000005 1234 4567        DC.W  $1234, $
8   8   000009 00                EVEN
    
```

## 9.2.22 LONGEVEN - Forcing Long-Word alignment

### Syntax

```
LONGEVEN
```

### Synonym

None

### Description

This directive forces the next instruction to the next long-word address relative to the start of the section. `LONGEVEN` is an abbreviation for `ALIGN 4`.

### Example

See the following listing for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

#### Listing: Forcing Long Word Alignment

```

2   2   000000 01                dcb.b 1,1
      ; location counter is not a multiple of 4; three filling
      ; bytes are required.
3   3   000001 0000 00                longeven
4   4   000004 0002 0002            dcb.w 2,2
      ; location counter is already a multiple of 4; no filling
      ; bytes are required.
5   5                                longeven
6   6   000008 0202                dcb.b 2,2
7   7   ; following is for text section
8   8                                s27                SECTION 27
    
```

```

9      9      000000 9D                      nop
          ; location counter is not a multiple of 4; three filling
          ; bytes are required.

10     10     000001 0000 00                longeven

11     11     000004 9D                      nop

```

## 9.2.23 MACRO - Begin macro definition

### Syntax

```
<label>: MACRO
```

### Synonym

None

### Description

The `<label>` of the `MACRO` directive is the name by which the macro is called. This name must not be a processor machine instruction or assembler directive name. For more information on macros, see the [Macros](#) chapter.

### Example

See the following listing for a macro definition.

#### Listing: Example macro definition

```

MyData: XDEF Start
        SECTION
char1:  DS.B 1
char2:  DS.B 1
cpChar: MACRO
        LDA  \1
        STA  \2
        ENDM
CodeSec: SECTION
Start:
        cpChar char1, char2

```

```
LDA    char1
STA    char2
```

## 9.2.24 MEXIT - Terminate Macro Expansion

### Syntax

```
MEXIT
```

### Synonym

None

### Description

`MEXIT` is usually used together with conditional assembly within a macro. In that case it may happen that the macro expansion should terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the [ENDM - End macro definition](#) directive.

### Example

The code in the following listing allows the replication of simple instructions or directives using `MACRO` with `MEXIT`.

#### Listing: Example assembly code using MEXIT

```
storage: XDEF  entry
          EQU  $00FF

save:    MACRO          ; Start macro definition

          LDX  #storage

          LDA  \1

          STA  0,x      ; Save first argument

          LDA  \2

          STA  2,x      ; Save second argument

          IFC  '\3', '' ; Is there a third argument?

              MEXIT    ; No, exit from macro

          ENDC

          LDA  \3      ; Save third argument

          STA  4,x
```

## Detailed descriptions of all assembler directives

```

                ENDM                ; End of macro definition

datSec: SECTION

char1: ds.b 1
char2: ds.b 1

codSec: SECTION

entry:

                save char1, char2

```

The following listing shows the macro expansion of the previous macro.

### Listing: Macro Expansion

Abs. Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----
1	1		XDEF entry
2	2	0000 00FF	storage: EQU \$00FF
3	3		
4	4		save: MACRO ; Start macro definition
5	5		LDX #storage
6	6		LDA \1
7	7		STA 0,x ; Save first arg
8	8		LDA \2
9	9		STA 2,x ; Save second arg
10	10		IFC '\3', '' ; is there a
11	11		MEXIT ; No, exit from macro.
12	12		ENDC
13	13		LDA \3 ; Save third argument
14	14		STA 4,X
15	15		ENDM ; End of macro defin
16	16		
17	17		datSec: SECTION
18	18	000000	char1: ds.b 1
19	19	000001	char2: ds.b 1
20	20		
21	21		
22	22		
23	23		codSec: SECTION

```

24 24          entry:
25 25          save char1, char2
26 5m 000000 AEFf  +      LDX #storage
27 6m 000002 C6 xxxx  +      LDA char1
28 7m 000005 E700  +      STA 0,x ; Save first arg
29 8m 000007 C6 xxxx  +      LDA char2
30 9m 00000A E702  +      STA 2,x ; Save second
31 10m          0000 0001  +      IFC '', '' ; Is there a
33 11m          +      MEXIT ; no, exit macro.
34 12m          +      ENDC
35 13m          +      LDA ; Save third argu
36 14m          +      STA 4,X
    
```

## 9.2.25 MLIST - List macro expansions

### Syntax

```
MLIST [ON|OFF]
```

### Description

When the `ON` keyword is entered with an `MLIST` directive, the Assembler includes the macro expansions in the listing and in the debug file.

When the `OFF` keyword is entered, the macro expansions are omitted from the listing and from the debug file.

This directive is not written to the listing and debug file, and the default value is `ON`.

### Synonym

None

### Example

The assembly code in the following listing, with `MLIST ON`, generates the assembler output listing in the next listing.

#### Listing: Example assembly source code using MLIST

```

XDEF entry
MLIST ON
    
```

## Detailed descriptions of all assembler directives

```

swap:  MACRO

        LDA   \1

        LDX   \2

        STA   \2

        STX   \1

        ENDM

codSec: SECTION

entry:

        LDA   #$F0

        LDX   #$0F

main:

        STA   first

        STX   second

        swap first, second

        NOP

        BRA   main

datSec: SECTION

first:  DS.B  1

second: DS.B  1

```

The following listing shows the output of the example assembly source code using MLIST listed above:

### Listing: Assembler Output

```

1      1
3      3          swap:  XDEF  entry
                          MACRO

4      4          LDA   \1

5      5          LDX   \2

6      6          STA   \2

7      7          STX   \1

8      8          ENDM

9      9

10     10         codSec: SECTION

11     11         entry:

12     12     000000 A6F0          LDA   #$F0

13     13     000002 AE0F          LDX   #$0F

14     14         main:

```

```

15 15 000004 C7 xxxx          STA  first
16 16 000007 CF xxxx          STX  second
17 17                          swap first, second
18 4m 00000A C6 xxxx  +      LDA  first
19 5m 00000D CE xxxx  +      LDX  second
20 6m 000010 C7 xxxx  +      STA  second
21 7m 000013 CF xxxx  +      STX  first
22 18 000016 9D              NOP
23 19 000017 20EB          BRA  main
24 20
25 21                          datSec: SECTION
26 22 000000              first: DS.B 1
27 23 000001              second: DS.B 1
    
```

For the same code, with `MLIST OFF`, the listing file is as shown in the following listing:

### Listing: Assembler Output

```

Abs. Rel. Loc  Obj. code Source line
-----
 1   1          XDEF  entry
 3   3          swap:  MACRO
 4   4          LDA  \1
 5   5          LDX  \2
 6   6          STA  \2
 7   7          STX  \1
 8   8          ENDM
 9   9          codSec: SECTION
10  10          entry:
11  11 000000 A6F0          LDA  #$F0
12  12 000002 AE0F          LDX  #$0F
13  13          main:
14  14 000004 C7 xxxx          STA  first
15  15 000007 CF xxxx          STX  second
16  16          swap first, second
21  17 000016 9D              NOP
22  18 000017 20EB          BRA  main
    
```

## Detailed descriptions of all assembler directives

```

23  19                               datSec: SECTION
24  20  000000                       first:  DS.B 1
25  21  000001                       second: DS.B 1

```

The `MLIST` directive does not appear in the listing file. When a macro is called after a `MLIST ON`, it is expanded in the listing file. If the `MLIST OFF` is encountered before the macro call, the macro is not expanded in the listing file.

## 9.2.26 NOLIST - Disable Listing

### Syntax

```
NOLIST
```

### Synonym

```
NOL
```

### Description

Suppresses the printing of the following instructions in the assembly listing and debug file until a [LIST - Enable Listing](#) assembler directive is reached.

### Example

See the following listing for an example of using `LIST` and `NOLIST`.

### Listing: Examples of LIST and NOLIST

```

aaa:  NOP
      LIST

bbb:  NOP
      NOP
      NOLIST

ccc:  NOP
      NOP
      LIST

ddd:  NOP
      NOP

```



The listing above generates the listing file in the following listing:

**Listing: Assembler Output**

Assembler Abs. Rel.	Loc	Obj. code	Source line
1 1	000000	9D	aaa: NOP
2 2			
4 4	000001	9D	bbb: NOP
5 5	000002	9D	NOP
6 6			
12 12	000005	9D	ddd: NOP
13 13	000006	9D	NOP

**See also**

[LIST - Enable Listing](#) assembler directive

**9.2.27 NOPAGE - Disable Paging**

**Syntax**

NOPAGE

**Synonym**

None

**Description**

Disables pagination in the listing file. Program lines are listed continuously, without headings or top or bottom margins.

**9.2.28 OFFSET - Create absolute symbols**

**Syntax**

OFFSET <expression>

## Synonym

None

## Description

The `OFFSET` directive declares an offset section and initializes the location counter to the value specified in <expression>. The <expression> must be absolute and may not contain references to external, undefined or forward defined labels.

## Example

The following listing shows how the `OFFSET` directive can be used to access an element of a structure.

### Listing: Example assembly source code

```

6      6      OFFSET 0
7      7      000000      ID:      DS.B  1
8      8      000001      COUNT:   DS.W  1
9      9      000003      VALUE:   DS.L  1
10     10      0000 0007  SIZE:    EQU   *
11     11
12     12      DataSec: SECTION
13     13      000000      Struct:  DS.B  SIZE
14     14
15     15      CodeSec: SECTION
16     16      entry:
17     17      000003 CE xxxx      LDX   #Struct
18     18      000006 8600      LDA   #0
19     19      000008 6A00      STA   ID, X
20     20      00000A 6201      INC   COUNT, X
21     21      00000C 42      INCA
22     22      00000D 6A03      STA   VALUE, X

```

When a statement affecting the location counter other than `EVEN`, `LONGEVEN`, `ALIGN`, or `DS` is encountered after the `OFFSET` directive, the offset section is ended. The preceding section is activated again, and the location counter is restored to the next available location in this section. The following listing shows the example where the location counter is changed.

### Listing: Example where the location counter is changed

```

7   7           ConstSec: SECTION
8   8   000000 11   cst1:   DC.B   $11
9   9   000001 13   cst2:   DC.B   $13
10  10
11  11           OFFSET 0
12  12   000000           ID:   DS.B   1
13  13   000001           COUNT: DS.W   1
14  14   000003           VALUE: DS.L   1
15  15           0000 0007  SIZE:  EQU   *
16  16
17  17   000002 22   cst3:   DC.B   $22
    
```

In the example above, the `cst3` symbol, defined after the `OFFSET` directive, defines a constant byte value. This symbol is appended to the section `ConstSec`, which precedes the `OFFSET` directive.

## 9.2.29 ORG - Set Location Counter

### Syntax

```
ORG <expression>
```

### Synonym

None

### Description

The `ORG` directive sets the location counter to the value specified by `<expression>`. Subsequent statements are assigned memory locations starting with the new location counter value. The `<expression>` must be absolute and may not contain any forward, undefined, or external references. The `ORG` directive generates an internal section, which is absolute (see the [Sections](#) chapter).

### Example

See the following listing for an example where `ORG` sets the location counter.

### Listing: Using ORG to set the location counter

```

        org    $2000
b1:     nop
b2:     rts

```

Viewing the following listing, you can see that the `b1` label is located at address \$2000 and label `b2` is at address \$2001.

### Listing: Assembler Output

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			org \$2000
2	2	a002000	9D	b1: nop
3	3	a002001	81	b2: rts

### See also

Assembler directives:

- [DC - Define Constant](#)
- [DCB - Define Constant Block](#)
- [DS - Define Space](#)
- [SECTION - Declare Relocatable Section](#)

## 9.2.30 PAGE - Insert Page break

### Syntax

PAGE

### Synonym

None

### Description

Insert a page break in the assembly listing.

### Example

The portion of code in the following listing demonstrates the use of a page break in the assembler output listing.

### Listing: Example assembly source code

```
code: SECTION
DC.B $00,$12

DC.B $00,$34

PAGE

DC.B $00,$56

DC.B $00,$78
```

The effect of the PAGE directive can be seen in the following listing.

**Listing: Assembler Output**

Abs.	Rel.	Loc	Obj.	code	Source line
----	----	-----	-----	-----	-----
1	1				code: SECTION
2	2	000000	0012		DC.B \$00,\$12
3	3	000002	0034		DC.B \$00,\$34
Abs.	Rel.	Loc	Obj.	code	Source line
----	----	-----	-----	-----	-----
5	5	000004	0056		DC.B \$00,\$56
6	6	000006	0078		DC.B \$00,\$78

**9.2.31 PLEN - Set Page Length**

**Syntax**

```
PLEN<n>
```

**Synonym**

None

**Description**

Sets the listings page length to <n> lines. <n> may range from 10 to 10000. If the number of lines already listed on the current page is greater than or equal to <n>, listing will continue on the next page with the new page length setting.

The default page length is 65 lines.

## 9.2.32 RAD50 - RAD50-encoded string constants

### Syntax

```
RAD50 <str>[, cnt]
```

### Synonym

None

### Description

This directive places strings encoded with the RAD50 encoding into constants. The RAD50 encoding places 3 string characters out of a reduced character set into 2 bytes. It therefore saves memory when comparing it with a plain ASCII representation. It also has some drawbacks, however. Only 40 different character values are supported, and the strings have to be decoded before they can be used. This decoding does include some computations including divisions (not just shifts) and is therefore rather expensive.

The encoding takes three bytes and looks them up in a string table. The following listing shows the RAD50 encoding.

### Listing: RAD50 Encoding

```
unsigned short LookUpPos(char x) {
    static const char translate[] =
        " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
    const char* pos = strchr(translate, x);
    if (pos == NULL) { EncodingError(); return 0; }
    return pos - translate;
}

unsigned short Encode(char a, char b, char c) {
    return LookUpPos(a) * 40 * 40 + LookUpPos(b) * 40
        + LookUpPos(c);
}
```

If the remaining string is shorter than 3 bytes, it is filled with spaces (which correspond to the RAD50 character 0).

The optional argument `cnt` can be used to explicitly state how many 16-bit values should be written. If the string is shorter than  $3 * cnt$ , then it is filled with spaces.

See the example C code below about how to decode it.

## Example

The string data in the following listing assembles to the following data where 11 characters are contained in eight bytes. The 11 characters in the string are represented by 8 bytes.

### Listing: RAD50 Example

```
XDEF rad50, rad50Len
DataSection SECTION

rad50:      RAD50 "Hello World"
rad50Len:   EQU (*-rad50)/2
```

### Listing: Assembler output where 11 characters are contained in eight bytes

```
$32D4 $4D58 $922A $4BA0
```

This C code shown in the following listing takes the data and prints "Hello World".

### Listing: Example-Program that Prints Hello World

```
#include "stdio.h"
extern unsigned short rad50[];

extern int rad50Len; /* address is value. Exported asm label */
#define rad50len ((int) &rad50Len)

void printRadChar(char ch) {
    static const char translate[] =
        " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
    char asciiChar = translate[ch];
    (void) putchar(asciiChar);
}

void PrintHallo(void) {
    unsigned char values = rad50len;
    unsigned char i;
    for (i=0; i < values; i++) {
        unsigned short val = rad50[i];
        printRadChar(val / (40 * 40));
        printRadChar((val / 40) % 40);
        printRadChar(val % 40);
    }
}
```

}

## 9.2.33 SECTION - Declare Relocatable Section

### Syntax

```
<name>: SECTION [SHORT] [<number>]
```

### Synonym

None

### Description

This directive declares a relocatable section and initializes the location counter for the following code. The first `SECTION` directive for a section sets the location counter to zero. Subsequent `SECTION` directives for that section restore the location counter to the value that follows the address of the last code in the section.

`<name>` is the name assigned to the section. Two `SECTION` directives with the same name specified refer to the same section.

`<number>` is optional and is only specified for compatibility with the MASM Assembler.

A section is a code section when it contains at least one assembly instruction. It is considered to be a constant section if it contains only `DC` or `DCB` directives. A section is considered to be a data section when it contains at least a `DS` directive or if it is empty.

### Example

The example in the following listing demonstrates the definition of a section `aaa`, which is split into two blocks, with section `bbb` in between them.

The location counter associated with the label `zz` is 1, because a `NOP` instruction was already defined in this section at label `xx`.

### Listing: Example of the SECTION assembler directive

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			aaa: SECTION 4
2	2	000000	9D	xx: NOP
3	3			bbb: SECTION 5



```

4   4   000000 9D           yy:    NOP
5   5   000001 9D           NOP
6   6   000002 9D           NOP
7   7           aaa:    SECTION 4
8   8   000001 9D           zz:    NOP
    
```

The optional qualifier `SHORT` specifies that the section is a short section, That means than the objects defined there can be accessed using the direct addressing mode.

For RS08, there are two additional section qualifiers: `RS08_SHORT` and `RS08_TINY`. When a section is declared as `RS08_SHORT` (or `RS08_TINY`) all the objects defined there can be accessed using the short (and respectively tiny) addressing modes.

The example in the following listing demonstrates the definition and usage of a `SHORT` section, and uses the direct addressing mode to access the symbol data.

### Listing: Using the direct addressing mode

```

1   1           dataSec: SECTION SHORT
2   2   000000           data:  DS.B 1
3   3
4   4           codeSec: SECTION
5   5
6   6           entry:
7   7   000000 9C           RSP
8   8   000001 A600         LDA  #0
9   9   000003 B7xx         STA  data
    
```

### See also

Assembler directives:

- [ORG - Set Location Counter](#)
- [DC - Define Constant](#)
- [DCB - Define Constant Block](#)
- [DS - Define Space](#)

## 9.2.34 SET - Set Symbol Value

### Syntax

<label>: SET <expression>

## Synonym

None

## Description

Similar to the [EQU - Equate symbol value](#) directive, the SET directive assigns the value of the <expression> in the operand field to the symbol in the <label> field. The <expression> must resolve as an absolute expression and cannot include a symbol that is undefined or not yet defined. The <label> is an assembly time constant. SET does not generate any machine code.

The value is temporary; a subsequent SET directive can redefine it.

## Example

See the following listing for examples of the SET directive.

### Listing: Using the SET assembler directive

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1		0000 0002	count: SET 2
2	2	000000	02	one: DC.B count
3	3			
4	4		0000 0001	count: SET count-1
5	5	000001	01	DC.B count
6	6			
7	7		0000 0001	IFNE count
8	8		0000 0000	count: SET count-1
9	9			ENDIF
10	10	000002	00	DC.B count

The value associated with the label `count` is decremented after each `DC.B` instruction.

## 9.2.35 SPC - Insert Blank Lines

### Syntax

```
SPC<count>
```

### Synonym

None

### Description

Inserts <count> blank lines in the assembly listing. <count> may range from 0 to 65. This has the same effect as writing that number of blank lines in the assembly source. A blank line is a line containing only a carriage return.

## 9.2.36 TABS - Set Tab Length

### Syntax

```
TABS <n>
```

### Synonym

None

### Description

Sets the tab length to <n> spaces. The default tab length is eight. <n> may range from 0 to 128.

## 9.2.37 TITLE - Provide Listing Title

### Syntax

```
TITLE "title"
```

### Synonym

TTL

### Description

Print the <title> on the head of every page of the listing file. This directive must be the first source code line. A title consists of a string of characters enclosed in quotes (").

The title specified will be written on the top of each page in the assembly listing file.

## 9.2.38 XDEF - External Symbol Definition

### Syntax

```
XDEF [.<size>] <label>[,<label>]...
```

where

```
<size> = B(direct), W (default),  
L or S or T
```

### Synonym

```
GLOBAL, PUBLIC
```

### Description

This directive specifies labels defined in the current module that are to be passed to the linker as labels that can be referenced by other modules linked to the current module.

The number of symbols enumerated in an XDEF directive is only limited by the memory available at assembly time.

The S and T size designators are only available for RS08, and result in marking the symbol as short or tiny.

### Example

See the following listing for the case where the XDEF assembler directive can specify symbols that can be used by other modules.

#### Listing: Using XDEF to create a variable to be used in another file

```
XDEF Count, main  
;; variable Count can be referenced in other modules,  
  
;; same for label main. Note that Linker & Assembler  
  
;; are case-sensitive, i.e., Count != count.
```

```
Count: DS.W 2
code: SECTION
main: DC.B 1
```

## 9.2.39 XREF - External Symbol Reference

### Syntax

```
XREF [<size>] <symbol>[,<symbol>]...
```

where <size> = B(direct ), W (default), or L or S or T.

### Synonym

```
EXTERNAL
```

### Description

This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 32-bit values is passed to the linker.

The number of symbols enumerated in an XREF directive is only limited by the memory available at assembly time.

The S and T size designators are only available for RS08, and result in marking the symbol as short or tiny.

### Example

```
XREF OtherGlobal ; Reference "OtherGlobal" defined in
                  ; another module. (See the XDEF
                  ; directive example.)
```

## 9.2.40 XREFB - External Reference for Symbols located on the Direct Page

### Syntax

```
XREFB <symbol>[, <symbol>] . . .
```

## Synonym

None

## Description

This directive specifies symbols referenced in the current module but defined in another module. Symbols enumerated in a `XREFB` directive, can be accessed using the direct address mode. The list of symbols and corresponding 8-bit values is passed to the linker.

The number of symbols enumerated in a `XREFB` directive is only limited by the memory available at assembly time.

## Example

```
XREFB OtherDirect ; Reference "OtherDirect" def in another  
                ; module (See XDEF directive example.)
```

## Chapter 10

# Macros

A macro is a template for a code sequence. Once a macro is defined, subsequent reference to the macro name are replaced by its code sequence.

### 10.1 Macro overview

A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently called.

The Assembler expands the macro definition each time the macro is called. The macro call causes source statements to be generated, which may include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Calling previously defined macros is also allowed. Source statements generated by a macro call are inserted in the source file at the position where the macro is invoked.

To call a macro, write the macro name in the operation field of a source statement. Place the arguments in the operand field. The macro may contain conditional assembly directives that cause the Assembler to produce in-line-coding variations of the macro definition.

Macros call produces in-line code to perform a predefined function. Each time the macro is called, code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program.

### 10.2 Defining a macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.

## Calling macros

- The body of the macro, a sequential list of assembler statements, some possibly including argument placeholders.
- The `ENDM` directive, terminating the macro definition.
- eventually an instruction `MEXIT`, which stops macro expansion.

See *the Assembler Directives* chapter for information about the `MACRO`, `ENDM`, `MEXIT`, and `MLIST` directives.

The body of a macro is a sequence of assembler source statements. Macro parameters are defined by the appearance of parameter designators within these source statements. Valid macro definition statements includes the set of processor assembly language instructions, assembler directives, and calls to previously defined macros. However, macro definitions may not be nested.

## 10.3 Calling macros

The form of a macro call is:

```
[<label>:] <name>[.<sizearg>] [<argument> [,<argument>]...]
```

Although a macro may be referenced by another macro prior to its definition in the source module, a macro must be defined before its first call. The name of the called macro must appear in the operation field of the source statement. Arguments are supplied in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the call, according to the macro definition and the arguments specified in the macro call. The source statements of the expanded macro are then assembled subject to the same conditions and restrictions affecting any source statement. Nested macros calls are also expanded at this time.

## 10.4 Macro parameters

As many as 36 different substitutable parameters can be used in the source statements that constitute the body of a macro. These parameters are replaced by the corresponding arguments in a subsequent call to that macro.



A parameter designator consists of a backslash character (\), followed by a digit (0 - 9) or an uppercase letter (A - Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Consider the following macro definition:

```
MyMacro: MACRO

    DC.\0    \1, \2

ENDM
```

When this macro is used in a program, e.g.:

```
MyMacro.B $10, $56
```

the Assembler expands it to:

```
DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator \1 through \9 and \A through \Z, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument itself is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

## 10.4.1 Macro argument grouping

To pass text including commas as a single macro argument, the Assembler supports a special syntax. This grouping starts with the [? ?] prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, escape brackets, question marks and backward slashes with a backward slash as prefix.

### NOTE

This escaping only takes place inside of [? ?] arguments. A backslash is only removed in this process if it is just before a bracket ([ ]), a question mark (?), or a second backslash (\).

#### Listing: Example macro definition

```
MyMacro:  MACRO
          DC      \1

          ENDM

MyMacro1: MACRO
          \1

          ENDM
```

The following listing shows the macro calls with rather complicated arguments:

#### Listing: Macro calls

```
MyMacro [?$10, $56?]
MyMacro ["\[?"]

MyMacro1 [?MyMacro [?$10, $56?]]
MyMacro1 [?MyMacro \[?$10, $56\?]]
```

These macro calls expand to the following listing:

#### Listing: Macro expansion

```
DC      $10, $56
DC      "[?"]

DC      $10, $56

DC      $10, $56
```

The Macro Assembler does also supports for compatibility with previous version's macro grouping with an angle bracket syntax, as in the following listing:

#### Listing: Angle bracket syntax

```
MyMacro <$10, $56>
```

However, this old syntax is ambiguous as `<` and `>` are also used as compare operators. For example, the following code does not produce the expected result:

### Listing: Potential problem using the angle-bracket syntax

```
MyMacro <1 > 2, 2 > 3> ; Wrong!
```

Because of this the old angle brace syntax should be avoided in new code. There is also an option to disable it explicitly.

See also the [-CMacBrackets: Square brackets for macro arguments grouping](#) and the [-CMacAngBrack: Angle brackets for grouping Macro Arguments](#) assembler options.

## 10.5 Labels inside macros

To avoid the problem of multiple-defined labels resulting from multiple calls to a macro that has labels in its source statements, the programmer can direct the Assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form `_nnnnn` where `nnnnn` is a 5-digit value. The programmer requests an assembler-generated label by specifying `\@` in a label field within a macro body. Each successive label definition that specifies a `\@` directive generates a successive value of `_nnnnn`, thereby creating a unique label on each macro call. Note that `\@` may be preceded or followed by additional characters for clarity and to prevent ambiguity.

The following listing shows the definition of the `clear` macro:

### Listing: Clear macro definition

```
clear:  MACRO
        LDX    #\1
        LDA    #16
\@LOOP: CLR    0,X
        INCX
        DECA
        BNE   \@LOOP
```

ENDM

This macro is called in the application, as listed in the following listing:

**Listing: Calling the clear macro**

```
clear    temporary
clear    data
```

The two macro calls of `clear` are expanded in the following manner, as listed in the following listing:

**Listing: Macro call expansion**

```
clear temporary
LDX    #temporary

LDA    #16

_00001LOOP:  CLR    0,X

INCX

DECA

BNE    _00001LOOP

clear data
LDX    #data

LDA    #16

_00002LOOP:  CLR    0,X

INCX

DECA

BNE    _00002LOOP
```

## 10.6 Macro expansion

When the Assembler reads a statement in a source program calling a previously defined macro, it processes the call as described in the following paragraphs.

The symbol table is searched for the macro name. If it is not in the symbol table, an undefined symbol error message is issued.

The rest of the line is scanned for arguments. Any argument in the macro call is saved as a literal or null value in one of the 35 possible parameter fields. When the number of arguments in the call is less than the number of parameters used in the macro the argument, which have not been defined at invocation time are initialize with "" (empty string).

Starting with the line following the `MACRO` directive, each line of the macro body is saved and is associated with the named macro. Each line is retrieved in turn, with parameter designators replaced by argument strings or assembler-generated label strings.

Once the macro is expanded, the source lines are evaluated and object code is produced.

## 10.7 Nested macros

Macro expansion is performed at invocation time, which is also the case for nested macros. If the macro definition contains nested macro call, the nested macro expansion takes place in line. Recursive macro calls are also supported.

A macro call is limited to the length of one line, i.e., 1024 characters.



## Chapter 11

# Assembler Listing File

The assembly listing file is the output file of the Assembler that contains information about the generated code. The listing file is generated when the `-L` assembler option is activated. When an error is detected during assembling from the file, no listing file is generated.

The amount of information available depends upon the following assembler options:

- `-L`: Generate a listing file
- `-Lc`: No Macro call in listing file
- `-Ld`: No macro definition in listing file
- `-Le`: No Macro expansion in listing file
- `-Li`: No included file in listing file

The information in the listing file also depends on following assembler directives:

- `LIST` - Enable Listing
- `NOLIST` - Disable Listing
- `CLIST` - List conditional assembly
- `MLIST` - List macro expansions

The format from the listing file is influenced by the following assembler directives:

- `PLEN` - Set Page Length
- `LLEN` - Set Line Length
- `TABS` - Set Tab Length
- `SPC` - Insert Blank Lines
- `PAGE` - Insert Page break
- `NOPAGE` - Disable Paging
- `TITLE` - Provide Listing Title.

The name of the generated listing file is `<base name>.lst`.

## 11.1 Page header

The page header consists of three lines:

- The first line contains an optional user string defined in the `TITLE` directive.  
The second line contains the name of the Assembler vendor (`Freescale`) as well as the target processor name - HC(S)08.
- The third line contains a copyright notice.

### Listing: Example page header output

```
Demo Application
Freescale HC08-Assembler
(c) COPYRIGHT Freescale 1991-2005
```

## 11.2 Source listing

The printed columns can be configured in various formats with the [-Lasmc: Configure listing file](#) assembler option. The default format of the source listing has the following five columns:

- [Abs.](#)
- [Rel.](#)
- [Loc](#)
- [Obj. code](#)
- [Source line](#)

### 11.2.1 Abs.

This column contains the absolute line number for each instruction. The absolute line number is the line number in the debug listing file, which contains all included files and where any macro calls have been expanded.

### Listing: Example output listing - Abs. column

```
Abs .
```



Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----
1			
	1		;-----
2			
	2		; File: test.o
3			
	3		;-----
4			
	4		
5			
	5		XDEF Start
6			
	6		MyData: SECTION
7			
	7	000000	char1: DS.B 1
8			
	8	000001	char2: DS.B 1
9			
	9		INCLUDE "macro.inc"
10			
	1i		cpChar: MACRO
11			
	2i		LDA \1
12			
	3i		STA \2
13			
	4i		ENDM
14			

**Source listing**

```

10          CodeSec: SECTION

15

11          Start:

16

12          cpChar char1, char2

17

2m 000000 C6 xxxx  +      LDA char1

18

3m 000003 C7 xxxx  +      STA char2

19

13 000006 9D          NOP

20

14 000007 9D          NOP

```

### 11.2.2 Rel.

This column contains the relative line number for each instruction. The relative line number is the line number in the source file. For included files, the relative line number is the line number in the included file. For macro call expansion, the relative line number is the line number of the instruction in the macro definition. See the listing below.

An *i* suffix is appended to the relative line number when the line comes from an included file. An *m* suffix is appended to the relative line number when the line is generated by a macro call.

**Listing: Example listing file - Rel. column**

```

Abs.
Rel.

  Loc   Obj. code   Source line
  ----
  ----
  -----
  -----

1      1
1

```

```

;-----
2
2
; File: test.o
3
3
;-----
4
4
5
5
XDEF Start
6
6
MyData: SECTION
7
7
000000 char1: DS.B 1
8
8
000001 char2: DS.B 1
9
9
INCLUDE "macro.inc"
10
10
1i
cpChar: MACRO
11
11
2i
LDA \1
12
12
3i
STA \2
13
13
4i
ENDM
14
10
CodeSec: SECTION
15
11
Start:

```

## Source listing

```

16
12                                     cpChar char1, char2

17
2m                                     LDA char1
000000 C6 xxxx +

18
3m                                     STA char2
000003 C7 xxxx +

19
13                                     NOP
000006 9D

20
14                                     NOP
000007 9D

```

In the previous example, the line number displayed in the `Rel.` column. represent the line number of the corresponding instruction in the source file.

`1i` on absolute line number `10` denotes that the instruction `cpChar: MACRO` is located in an included file.

`2m` on absolute line number `17` denotes that the instruction `LDA char1` is generated by a macro expansion.

### 11.2.3 Loc

This column contains the address of the instruction. For absolute sections, the address is preceded by an `a` and contains the absolute address of the instruction. For relocatable sections, this address is the offset of the instruction from the beginning of the relocatable section. This offset is a hexadecimal number coded on 6 digits.

A value is written in this column in front of each instruction generating code or allocating storage. This column is empty in front of each instruction that does not generate code (for example `SECTION`, `XDEF`). See the following listing:

#### Listing: Example Listing File - Loc column

```

Abs. Rel.
Loc
Obj. code  Source line
-----

```

```

1  1          ;-----
2  2          ; File: test.o
3  3          ;-----
4  4
5  5          XDEF  Start
6  6          MyData: SECTION
7  7
000000
          char1:  DS.B  1
8  8
000001
          char2:  DS.B  1
9  9          INCLUDE "macro.inc"
10 1i         cpChar:  MACRO
11 2i         LDA   \1
12 3i         STA   \2
13 4i         ENDM
14 10        CodeSec: SECTION
15 11        Start:
16 12        cpChar char1, char2
17 2m
000000
C6 xxxx  +          LDA  char1
18 3m
000003
C7 xxxx  +          STA  char2
19 13
000006
9D          NOP
20 14
000007
9D          NOP
    
```

In the previous example, the hexadecimal number displayed in the column `Loc.` is the offset of each instruction in the section `codeSec`.

There is no location counter specified in front of the instruction `INCLUDE"macro.inc"` because this instruction does not generate code.

The instruction `LDA char1` is located at offset 0 from the section `codeSec` start address.

## Source listing

The instruction `STA char2` is located at offset 3 from the section `codeSec` start address.

### 11.2.4 Obj. code

This column contains the hexadecimal code of each instruction in hexadecimal format. This code is not identical to the code stored in the object file. The letter ``x'` is displayed at the position where the address of an external or relocatable label is expected. Code at any position when ``x'` is written will be determined at link time. See the following listing:

#### Listing: Example listing file - Obj. code column

```

Abs. Rel.  Loc
Obj. code

Source line
-----
1  1          ;-----
2  2          ; File: test.o
3  3          ;-----
4  4
5  5          XDEF Start
6  6          MyData: SECTION
7  7  000000   char1: DS.B 1
8  8  000001   char2: DS.B 1
9  9          INCLUDE "macro.inc"
10 1i         cpChar: MACRO
11 2i         LDA  \1
12 3i         STA  \2
13 4i         ENDM
14 10        CodeSec: SECTION
15 11        Start:
16 12        cpChar char1, char2
17 2m 000000
C6 xxxxx
+          LDA char1
18 3m 000003
C7 xxxxx
+          STA char2

```

```

19      13  000006
9D
                                NOP

20      14  000007
9D
                                NOP

```

## 11.2.5 Source line

This column contains the source statement. This is a copy of the source line from the source module. For lines resulting from a macro expansion, the source line is the expanded line, where parameter substitution has been done. See the following listing:

### Listing: Example listing file - Source line column

```

Abs. Rel.  Loc   Obj. code
Source line
-----
1      1
;-----

2      2
; File: test.o

3      3
;-----

4      4
5      5
XDEF Start

6      6
MyData: SECTION

7      7  000000
char1: DS.B 1

8      8  000001
char2: DS.B 1

```

## Source listing

```

9      9
INCLUDE "macro.inc"

```

```

10     1i
cpChar: MACRO

```

```

11     2i
LDA   \1

```

```

12     3i
STA   \2

```

```

13     4i
ENDM

```

```

14     10
CodeSec: SECTION

```

```

15     11
Start:

```

```

16     12
cpChar char1, char2

```

```

17     2m 000000 C6 xxxx  +
LDA char1

```

```

18     3m 000003 C7 xxxx  +
STA char2

```

```

19     13 000006 9D
NOP

```

```

20     14 000007 9D
NOP

```



## Chapter 12

# Mixed C and Assembler Applications

To build mixed C and Assembler applications, you have to know how the C Compiler uses registers and calls procedures. The following sections will describe this for compatibility with the compiler. If you are working with another vendor's ANSI-C compiler, refer to your Compiler Manual to get the information about parameter passing rules.

When you intend to mix Assembly source file and ANSI-C source files in a single application, the following issues are important:

- [Memory models](#)
- [Parameter passing scheme](#)
- [Return Value](#)
- [Accessing assembly variables in an ANSI-C source file](#)
- [Accessing ANSI-C variables in an assembly source file](#)
- [Invoking an assembly function in an ANSI-C source file](#)
- [Support for structured types](#)
- [Structured type: Limitations](#)

### 12.1 Memory models

The memory models are only important if you mix C and assembly code. In this case all sources must be compiled or assembled with the same memory model.

The Assembler supports all memory models of the compiler. Depending on your hardware, use the smallest memory model suitable for your programming needs.

The following table summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

**Table 12-1. HC08 memory models**

Option	Memory Model	Local Data	Global Data	Suggested Use
-Ms	SMALL	SP rel	extended	The SMALL memory model is the default. All pointers and functions are assumed to have 16-bit addresses if not explicitly specified. In the SMALL memory model, code and data must be in the 64k address space.
-Mt	TINY	SP rel	direct	In the TINY memory model, all data including stack must fit into the zero page. Data pointers are assumed to have 8-bit addresses if not explicitly specified with the keyword <code>__far</code> . The code address space is still 64k and function pointers are still 16 bits in length.

**NOTE**

The default pointer size for the compiler is also affected by the memory model chosen.

## 12.2 Parameter passing scheme

Check the backend chapter in the compiler manual for the details of parameter passing.

## 12.3 Return Value

Check the backend chapter in the compiler manual for the details of parameter passing.

## 12.4 Accessing assembly variables in an ANSI-C source file

A variable or constant defined in an assembly source file is accessible in an ANSI-C source file.

The variable or constant is defined in the assembly source file using the standard assembly syntax.

Variables and constants must be exported using the `XDEF` directive to make them visible from other modules, as listed in the following listing:

### Listing: Example of data and constant definition

```
                XDEF  ASMData, ASMConst
DataSec:  SECTION

ASMData:  DS.W  1          ; Definition of a variable

ConstSec: SECTION

ASMConst: DC.W  $44A6    ; Definition of a constant
```

We recommend that you generate a header file for each assembler source file. This header file should contain the interface to the assembly module.

An external declaration for the variable or constant must be inserted in the header file, as listed in the following listing:

### Listing: Example of data and constant declarations

```
/* External declaration of a variable */
extern int      ASMData;

/* External declaration of a constant */
extern const int ASMConst;
```

The variables or constants can then be accessed in the usual way, using their names, as listed in the following listing:

### Listing: Example of data and constant reference

```
ASMData = ASMConst + 3;
```

## 12.5 Accessing ANSI-C variables in an assembly source file

## Invoking an assembly function in an ANSI-C source file

A variable or constant defined in an ANSI-C source file is accessible in an assembly source file.

The variable or constant is defined in the ANSI-C source file using the standard ANSI-C syntax, as listed in the following listing:

### Listing: Example definition of data and constants

```
unsigned int CData;          /* Definition of a variable */
unsigned const int CConst; /* Definition of a constant */
```

An external declaration for the variable or constant must be inserted into the assembly source file, as listed in the following listing:

This can also be done in a separate file, included in the assembly source file.

### Listing: Example declaration of data and constants

```
XREF CData; External declaration of a variable
XREF CConst; External declaration of a constant
```

The variables or constants can then be accessed in the usual way, using their names, as listed in the following listing:

### Listing: Example of data and constant reference

```
LDA CConst
....

LDA CData
....
```

#### NOTE

The compiler supports also the automatic generation of assembler include files. See the description of the `-La` compiler option in the compiler manual.

## 12.6 Invoking an assembly function in an ANSI-C source file

An function implemented in an assembly source file (`mixasm.asm` in the listing *Example of an assembly file: mixasm.asm*) can be invoked in a C source file ([Listing: Example C source code file: mixc.c](#)). During the implementation of the function in the assembly source file, you should pay attention to the parameter passing scheme of the ANSI-C compiler you are using in order to retrieve the parameter from the right place.

**Listing: Example of an assembly file: mixasm.asm**

```

        XREF CData
        XDEF AddVar

        XDEF ASMData

DataSec: SECTION

ASMData: DS.B 1

CodeSec: SECTION

AddVar:

        ADD CData    ; add CData to the parameter in register A
        STA ASMData ; result of the addition in ASMData

        RTS

```

We recommend that you generate a header file for each assembly source file, as listed in the above listing. This header file (`mixasm.h` in the listing *Header file for the assembly mixasm.asm file: mixasm.h*) should contain the interface to the assembly module.

**Listing: Header file for the assembly mixasm.asm file: mixasm.h**

```

/* mixasm.h */
#ifndef _MIXASM_H_
#define _MIXASM_H_

void AddVar(unsigned char value);

/* function that adds the parameter value to global CData */
/* and then stores the result in ASMData */
/* variable which receives the result of AddVar */

extern char ASMData;
#endif /* _MIXASM_H_ */

```

The function can then be invoked in the usual way, using its name.

**12.6.1 Example of a C file**

A C source code file (`mixc.c`) has the `main()` function which calls the `AddVar()` function. See the following listing. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format.)

**Listing: Example C source code file: mixc.c**

## Invoking an assembly function in an ANSI-C source file

```

static int Error          = 0;
const unsigned char CData = 12;

#include "mixasm.h"

void main(void) {

    AddVar(10);

    if (ASMDData != CData + 10){

        Error = 1;

    } else {

        Error = 0;

    }

    for(;;); // wait forever

}

```

### CAUTION

Be careful, as the Assembler will not make any checks on the number and type of the function parameters.

The application must be correctly linked.

For these `c` and `*.asm` files, a possible linker parameter file is shown in the following listing.

### Listing: Example of linker parameter file: `mixasm.prm`

```

LINK mixasm.abs
NAMES

    mixc.o mixasm.o

END

SECTIONS

    MY_ROM    = READ_ONLY  0x4000 TO 0x4FFF;
    MY_RAM    = READ_WRITE 0x2400 TO 0x2FFF;
    MY_STACK = READ_WRITE 0x2000 TO 0x23FF;

END

PLACEMENT

    DEFAULT_RAM    INTO MY_RAM;
    DEFAULT_ROM    INTO MY_ROM;
    SSTACK         INTO MY_STACK;

END

INIT main

```

**NOTE**

We recommend that you use the same memory model and object file format for all the generated object files.

## 12.7 Support for structured types

When the [-Struct: Support for structured types](#) assembler option is activated, the Macro Assembler also supports the definition and usage of structured types. This allows an easier way to access ANSI-C structured variable in the Macro Assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type. See [Structured type definition](#).
- Define a structured variable. See [Variable definition](#).
- Declare a structured variable. See [Variable declaration](#).
- Access the address of a field inside of a structured variable. See [Accessing a field address](#)
- Access the offset of a field inside of a structured variable. See [Accessing a field offset](#).

**NOTE**

Some limitations apply in the usage of the structured types in the Macro Assembler. See [Structured type: Limitations](#).

### 12.7.1 Structured type definition

The Macro Assembler is extended with the following new keywords in order to support ANSI-C type definitions.

- STRUCT
- UNION

The structured type definition for `STRUCT` can be encoded as in the following listing:

**Listing: Definition for STRUCT**

```
typeName: STRUCT
  lab1: DS.W 1   lab2: DS.W 1   ...
```

ENDSTRUCT

where:

- `typeName` is the name associated with the defined type. The type name is considered to be a user-defined keyword. The Macro Assembler will be case-insensitive on `typeName`.
- `STRUCT` specifies that the type is a structured type.
- `lab1` and `lab2` are the fields defined inside of the `typeName` type. The fields will be considered as user-defined labels, and the Macro Assembler will be case-sensitive on label names.
- As with all other directives in the Assembler, the `STRUCT` and `UNION` directives are case-insensitive.
- The `STRUCT` and `UNION` directives cannot start on column 1 and must be preceded by a label.

## 12.7.2 Types allowed for structured type fields

The field inside of a structured type may be:

- another structured type or
- a base type, which can be mapped on 1, 2, or 4 bytes.

The following table lists the ANSI-C standard types and their converted equivalents in the assembler notation:

**Table 12-2. Converting ANSI-C standard types to assembler notation**

ANSI-C type	Assembler Notation
char	<a href="#">DS - Define Space</a>
short	DS.W
int	DS.W
long	DS.L
enum	DS.W
bitfield	-- not supported --
float	-- not supported --
double	-- not supported --
data pointer	DS.W
function pointer	-- not supported --



### 12.7.3 Variable definition

The Macro Assembler can provide a way to define a variable with a specific type. This is done using the following syntax ( *Listing: Assembly code analog of a C struct of type: myType*):

```
var: typeName
```

where:

- `var` is the name of the variable.
- `typeName` is the type associated with the variable.

**Listing: Assembly code analog of a C struct of type: myType**

```
myType:    STRUCT
field1:    DS.W 1

field2:    DS.W 1

field3:    DS.B 1

field4:    DS.B 3

field5:    DS.W 1

            ENDSTRUCT

DataSection: SECTION

structVar: TYPE myType ; var `structVar' is of type `myType'
```

### 12.7.4 Variable declaration

The Macro Assembler can provide a way to associated a type with a symbol which is defined externally. This is done by extending the `XREF` syntax:

```
XREF var: typeName, var2
```

where:

- `var` is the name of an externally defined symbol.
- `typeName` is the type associated with the variable `var`.

## Support for structured types

`var2` is the name of another externally defined symbol. This symbol is not associated with any type. See the following listing for an example.

### Listing: Example of extending XREF

```
myType: STRUCT
field1:  DS.W 1

field2:  DS.W 1

field3:  DS.B 1

field4:  DS.B 3

field5:  DS.W 1

        ENDSTRUCT

        XREF extData: myType ; var `extData' is type `myType'
```

## 12.7.5 Accessing a structured variable

The Macro Assembler can provide a means to access each structured type field absolute address and offset.

### 12.7.5.1 Accessing a field address

To access a structured-type field address (Listing: Example of accessing a field address), the Assembler uses the colon character ':'.

```
var:field
```

where

- `var` is the name of a variable, which was associated with a structured type.
- `field` is the name of a field in the structured type associated with the variable.

#### Listing: Example of accessing a field address

```
myType:  STRUCT
field1:  DS.W 1

field2:  DS.W 1

field3:  DS.B 1

field4:  DS.B 3
```

```

field5:    DS.W 1

          ENDSTRUCT

          XREF  myData:myType

          XDEF  entry

CodeSec: SECTION

entry:

          LDA  myData:field3 ; Loads register A with the content of
                               ; field field3 from variable myData.

```

**NOTE**

The period cannot be used as separator because in assembly language it is a valid character inside of a symbol name.

**12.7.5.2 Accessing a field offset**

To access a structured type field offset, the Assembler will use following notation:

```
<typeName>-><field>
```

where:

- `typeName` is the name of a structured type.
- `field` is the name of a field in the structured type associated with the variable. See the following listing for an example of using this notation for accessing an offset.

**Listing: Accessing a field offset with the -><field> notation**

```

myType:  STRUCT
field1:  DS.W 1

field2:  DS.W 1

field3:  DS.B 1

field4:  DS.B 3

field5:  DS.W 1

          ENDSTRUCT

          XREF.B myData

          XDEF  entry

CodeSec: SECTION

```

## Structured type: Limitations

entry:

```
LDX #myData
LDA myType->field3,X ; Adds the offset of field 'field3'
                    ; (4) to X and loads A with the
                    ; content of the pointed address
```

## 12.8 Structured type: Limitations

A field inside of a structured type may be:

- another structured type
- a base type, which can be mapped on 1, 2, or 4 bytes.

The Macro Assembler is not able to process bitfields or pointer types.

The type referenced in a variable definition or declaration must be defined previously. A variable cannot be associated with a type defined afterwards.

# Chapter 13

## Make Applications

This chapter has the following sections:

- [Assembly applications](#)
- [Memory maps and segmentation](#)

### 13.1 Assembly applications

This section covers:

- [Directly generating an absolute file](#)
- [Mixed C and assembly applications](#)

#### 13.1.1 Directly generating an absolute file

When an absolute file is directly generated by the Assembler:

- the application entry point must be specified in the assembly source file using the directive `ABSENTRY`.
- The whole application must be encoded in a single assembly unit.
- The application should only contain absolute sections.

##### 13.1.1.1 Generating object files

The entry point of the application must be mentioned in the Linker parameter file using the `INIT funcname` command. The application is build of the different object files with the Linker. The Linker is document in a separate document.

Your assembly source files must be separately assembled. Then the list of all the object files building the application must be enumerated in the application PRM file.

### 13.1.2 Mixed C and assembly applications

Normally the application starts with the main procedure of a C file. All necessary object files - assembly or C - are linked with the Linker in the same fashion like pure C applications. The Linker is documented in a separate document.

## 13.2 Memory maps and segmentation

Relocatable Code Sections are placed in the `DEFAULT_ROM` or `.text` Segment.

Relocatable Data Sections are placed in the `DEFAULT_RAM` or `.data` Segment.

#### NOTE

The `.text` and `.data` names are only supported when the ELF object file format is used.

There are no checks at all that variables are in `RAM`. If you mix code and data in a section you cannot place the section into `ROM`. That is why we suggest that you separate code and data into different sections.

If you want to place a section in a specific address range, you have to put the section name in the placement portion of the linker parameter file, as listed in the following listing:

#### Listing: Example assembly source code

```
SECTIONS
  ROM1      = READ_ONLY  0x0200 TO 0x0FFF;

  SpecialROM = READ_ONLY  0x8000 TO 0x8FFF;

  RAM       = READ_WRITE 0x4000 TO 0x4FFF;

PLACEMENT

  DEFAULT_ROM  INTO ROM1;
```

```
mySection      INTO SpecialROM;  
DEFAULT_RAM    INTO RAM;  
END
```





# Chapter 14

## How to...

This chapter covers the following topics:

- [Working with absolute sections](#)
- [Working with relocatable sections](#)
- [Initializing the Vector table](#)
- [Splitting an application into modules](#)
- [Using the direct addressing mode to access symbols](#)

### 14.1 Working with absolute sections

An absolute section is a section whose start address is known at assembly time.

(See modules `fiboorg.asm` and `fiboorg.prm` in the demo directory.)

#### 14.1.1 Defining absolute sections in an assembly source file

An absolute section is defined using the `ORG` directive. In that case, the Macro Assembler generates a pseudo section, whose name is "`ORG_<index>`", where `index` is an integer which is incremented each time an absolute section is encountered, as listed in the following listing:

**Listing: Defining an absolute section containing data**

```
var:    ORG    $800    ; Absolute data section.
        DS.    1
        ORG    $A00    ; Absolute constant data section.
cst1:  DC.B  $A6
```

## working with absolute sections

```
cst2: DC.B $BC
```

In the previous portion of code, the label `cst1` is located at address `$A00`, and label `cst2` is located at address `$A01`.

### Listing: Assembler Output

```

1      1                                ORG    $800
2      2  a000800      var:  DS.B  1

3      3                                ORG    $A00
4      4  a000A00 A6    cst1:  DC.B  $A6
5      5  a000A01 BC    cst2:  DC.B  $BC
```

Locate program assembly source code in a separate absolute section, as listed in the following listing:

### Listing: Defining an absolute section containing code

```

XDEF  entry
ORG   $C00 ; Absolute code section.

entry:

LDA   cst1 ; Load value in cst1
ADD   cst2 ; Add value in cst2

STA   var  ; Store in var

BRA   entry
```

In the portion of assembly code above, the `LDA` instruction is located at address `$C00`, and the `ADD` instruction is at address `$C03`. See the following listing:.

### Listing: Assembler Output

```

8      8                                ORG    $C00 ; Absolute code
9      9      entry:

10     10  a000C00 C6 0A00      LDA   cst1 ; Load value
11     11  a000C03 CB 0A01      ADD   cst2 ; Add value
12     12  a000C06 C7 0800      STA   var  ; Store in var
13     13  a000C09 20F5      BRA   entry

14     14
```

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used.
- The `RSP` instruction can be used to initialize the stack pointer to `$FF`.

- Publish the application's entry point using `XDEF`.
- The programmer should ensure that the addresses specified in the source files are valid addresses for the MCU being used.

## 14.1.2 Linking an application containing absolute sections

When the Assembler is generating an object file, applications containing only absolute sections must be linked. The linker parameter file must contain at least:

- the name of the absolute file
- the name of the object file which should be linked
- the specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- the specification of a memory area where the sections containing code or constants must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- the specification of the application entry point, and
- the definition of the reset vector.

The minimal linker parameter file will look as shown in the following listing:.

### Listing: Minimal linker parameter file

```
LINK test.abs /* Name of the executable file generated. */
NAMES

    test.o /* Name of the object file in the application. */

END

SECTIONS

/* READ_ONLY memory area. There should be no overlap between this
memory area and the absolute sections defined in the assembly
source file.
*/
MY_ROM = READ_ONLY 0x4000 TO 0x4FFF;

/* READ_WRITE memory area. There should be no overlap between this
memory area and the absolute sections defined in the assembly
source file.
*/
```

### working with relocatable sections

```

MY_RAM = READ_WRITE 0x2000 TO 0x2FFF;

END

PLACEMENT

/* Relocatable variable sections are allocated in MY_RAM. */
DEFAULT_RAM INTO MY_RAM;

/* Relocatable code and constant sections are allocated in MY_ROM. */
DEFAULT_ROM INTO MY_ROM;

END

INIT entry /* Application entry point. */
VECTOR ADDRESS 0xFFFFE entry /* Initialization of the reset vector. */

```

#### NOTE

There should be no overlap between the absolute sections defined in the assembly source file and the memory areas defined in the PRM file.

#### NOTE

As the memory areas (segments) specified in the PRM file are only used to allocate relocatable sections, nothing will be allocated there when the application contains only absolute sections. In that case you can even specify invalid address ranges in the PRM file.

## 14.2 Working with relocatable sections

A relocatable section is a section which start address is determined at linking time.

### 14.2.1 Defining relocatable sections in a source file

Define a relocatable section using the `SECTION` directive. See the following listing for an example of defining relocatable sections.

#### Listing: Defining relocatable sections containing data

```

constSec: SECTION ; Relocatable constant data section.
cst1: DC.B $A6

cst2: DC.B $BC

```

```

dataSec: SECTION    ; Relocatable data section.

var:      DS.B  1
    
```

In the previous portion of code, the label `cst1` will be located at an offset 0 from the section `constSec` start address, and label `cst2` will be located at an offset 1 from the section `constSec` start address. See the following listing:

### Listing: Assembler Output

```

2      2      constSec: SECTION ; Relocatable
3      3      000000 A6      cst1:      DC.B      $A6

4      4      000001 BC      cst2:      DC.B      $BC

5      5

6      6      dataSec: SECTION ; Relocatable

7      7      000000      var:      DS.B      1
    
```

Locate program assembly source code in a separate relocatable section, as listed in the following listing:

### Listing: Defining a relocatable section for code

```

XDEF entry
codeSec: SECTION    ; Relocatable code section.

entry:

LDA    cst1    ; Load value in cst1

ADD    cst2    ; Add value in cst2

STA    var     ; Store in var

BRA    entry
    
```

In the previous portion of code, the `LDA` instruction is located at an offset 0 from the `codeSec` section start address, and `ADD` instruction at an offset 3 from the `codeSec` section start address.

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used
- The `RSP` instruction can be used to initialize the stack pointer to `$FF`.
- Publish the application's entry point using the `XDEF` directive.

## 14.2.2 Linking an application containing relocatable sections

Applications containing relocatable sections must be linked. The linker parameter file must contain at least:

- the name of the absolute file,
- the name of the object file which should be linked,
- the specification of a memory area where the sections containing variables must be allocated,
- the specification of a memory area where the sections containing code or constants must be allocated,
- the specification of the application's entry point, and
- the definition of the reset vector.

A minimal linker parameter file will look as shown in the following listing:

**Listing: Minimal linker parameter file**

```

/* Name of the executable file generated.      */
LINK test.abs

/* Name of the object file in the application. */

NAMES

    test.o

END

SECTIONS

/* READ_ONLY memory area. */

    MY_ROM = READ_ONLY 0x2B00 TO 0x2BFF;

/* READ_WRITE memory area. */

    MY_RAM = READ_WRITE 0x2800 TO 0x28FF;

END

PLACEMENT

/* Relocatable variable sections are allocated in MY_RAM.      */

    DEFAULT_RAM          INTO MY_RAM;

/* Relocatable code and constant sections are allocated in MY_ROM. */

    DEFAULT_ROM, constSec INTO MY_ROM;

END

INIT entry              /* Application entry point.      */

VECTOR ADDRESS 0xFFFFE entry /* Initialization of the reset vector. */

```

**NOTE**

The programmer should ensure that the memory ranges he specifies in the SECTIONS block are valid addresses for the

controller he is using. In addition, when using the SDI debugger the addresses specified for code or constant sections must be located in the target board ROM area. Otherwise, the debugger will not be able to load the application

## 14.3 Initializing Vector table

The vector table can be initialized in the assembly source file or in the linker parameter file. We recommend that you initialize it in the linker parameter file.

- Initializing the Vector table in the linker PRM file (recommended),
- Initializing the Vector Table in a source file using a relocatable section, or
- Initializing the Vector Table in a source file using an absolute section.

The HC(S)08 allows 128 entries in the vector table starting at memory location `$FF00` extending to memory location `$FFFF`.

The Reset vector is located in `$FFFE`, and the SWI interrupt vector is located in `$FFFC`. From `$FFFA` down to `$FF00` are located the `IRQ[0]` interrupt (`$FFFA`), `IRQ[1]` (`$FFFA`),..., `IRQ[125]` (`$FF00`).

In the following examples, the Reset vector, the SWI interrupt and the `IRQ[1]` interrupt are initialized. The `IRQ[0]` interrupt is not used.

The topics covered here:

- [Initializing the Vector table in the linker PRM file](#)
- [Initializing the Vector Table in a source file using a relocatable section](#),
- [Initializing the Vector Table in a source file using an absolute section](#)

### 14.3.1 Initializing the Vector table in the linker PRM file

Initializing the vector table from the PRM file allows you to initialize single entries in the table. The user can decide to initialize all the entries in the vector table or not.

The labels or functions, which should be inserted in the vector table, must be implemented in the assembly source file (*Listing: Initializing the Vector table from a PRM File*). All these labels must be published, otherwise they cannot be addressed in the linker PRM file.

#### Listing: Initializing the Vector table from a PRM File

## Initializing Vector table

```

XDEF  IRQ1Func, SWIFunc, ResetFunc
DataSec: SECTION

Data:   DS.W 5           ; Each interrupt increments an element
                ; of the table.

CodeSec: SECTION
; Implementation of the interrupt functions.

IRQ1Func:
        LDA   #0
        BRA   int

SWIFunc:
        LDA   #4
        BRA   int

ResetFunc:
        LDA   #8
        BRA   entry

int:
        PSHH
        LDHX #Data ; Load address of symbol Data in X
; X <- address of the appropriate element in the tab

Offset:  TSTA
        BEQ  Offset3

Offset2:
        AIX  #1
        DECA
        BNE  Offset2

Offset3:
        INC  0, X ; The table element is incremented
        PULH
        RTI

entry:
        LDHX #0E00 ; Init Stack Pointer to $E00-$1=$DFF
        TXS
        CLRX
        CLRH
        CLI           ; Enables interrupts

```



```
loop:   BRA    loop
```

### NOTE

The `IRQ1Func`, `SWIFunc`, and `ResetFunc` functions are published. This is required, because they are referenced in the linker PRM file.

### NOTE

The HC08 processor automatically pushes the PC, X, A, and CCR registers on the stack when an interrupt occurs. The interrupt functions do not need to save and restore those registers. To maintain compatibility with the M6805 Family, the H register is not stacked. It is the user's responsibility to save and restore it prior to returning.

### NOTE

All Interrupt functions must be terminated with an `RTI` instruction

The vector table is initialized using the linker `VECTOR ADDRESS` command, as listed in the following listing:

### Listing: Using the VECTOR ADDRESS Linker Command

```
LINK test.abs
NAMES

    test.o

END

SECTIONS

    MY_ROM    = READ_ONLY    0x0800 TO 0x08FF;
    MY_RAM    = READ_WRITE   0x0B00 TO 0x0CFF;
    MY_STACK  = READ_WRITE   0x0D00 TO 0x0DFF;

END

PLACEMENT

    DEFAULT_RAM      INTO MY_RAM;
    DEFAULT_ROM      INTO MY_ROM;
    SSTACK           INTO MY_STACK;

END

INIT ResetFunc

VECTOR ADDRESS 0xFFFF8 IRQ1Func
VECTOR ADDRESS 0xFFFFC SWIFunc
VECTOR ADDRESS 0xFFFFE ResetFunc
```

**NOTE**

The statement `INIT ResetFunc` defines the application entry point. Usually, this entry point is initialized with the same address as the reset vector.

**NOTE**

The statement `VECTOR ADDRESS 0xFFFF8 IRQ1Func` specifies that the address of the `IRQ1Func` function should be written at address `0xFFFF8`.

### 14.3.2 Initializing the Vector Table in a source file using a relocatable section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions that should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See the following listing:

**Listing: Initializing the Vector Table in source code with a relocatable section**

```

XDEF  ResetFunc
XDEF  IRQ0Int

DataSec: SECTION

Data:   DS.W 5 ; Each interrupt increments an element of the table.

CodeSec: SECTION

; Implementation of the interrupt functions.

IRQ1Func:

    LDA  #0

    BRA  int

SWIFunc:

    LDA  #4

    BRA  int

ResetFunc:

    LDA  #8

    BRA  entry

```

```

DummyFunc:
    RTI

int:
    PSHH
    LDHX #Data ; Load address of symbol Data in X
    ; X <- address of the appropriate element in the tab

Offset:
    TSTA
    BEQ  Offset3

Offset2:
    AIX  #$1
    DECA
    BNE  Offset2

Offset3:
    INC  0, X ; The table element is incremented
    PULH
    RTI

entry:
    LDHX #$0E00 ; Init Stack Pointer to $E00-$1=$DFF
    TXS
    CLRX
    CLRH
    CLI ; Enables interrupts

loop:    BRA  loop

VectorTable: SECTION
; Definition of the vector table.
IRQ1Int: DC.W  IRQ1Func
IRQ0Int: DC.W  DummyFunc
SWIInt:   DC.W  SWIFunc
ResetInt: DC.W  ResetFunc

```

## NOTE

Each constant in the `VectorTable` section is defined as a word (a 2-byte constant), because the entries in the vector table are 16 bits wide.

### NOTE

In the previous example, the constant `IRQ1Int` is initialized with the address of the label `IRQ1Func`. The constant `IRQ0Int` is initialized with the address of the label `Dummy Func` because this interrupt is not in use.

### NOTE

All the labels specified as initialization value must be defined, published (using `XDEF`) or imported (using `XREF`) before the vector table section. No forward reference is allowed in the `DC` directive.

### NOTE

The constant `IRQ0Int` is exported so that the section containing the vector table is linked with the application.

The section should now be placed at the expected address. This is performed in the linker parameter file, as listed in the following listing:

#### Listing: Example linker parameter file

```
LINK test.abs
NAMES

    test.o+

END

ENTRIES

    IRQ0Int

END

SECTIONS

    MY_ROM    = READ_ONLY  0x0800 TO 0x08FF;
    MY_RAM    = READ_WRITE 0x0B00 TO 0x0CFF;
    MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;

/* Define the memory range for the vector table */

    Vector    = READ_ONLY  0xFFFF8 TO 0xFFFF;

END

PLACEMENT

    DEFAULT_RAM      INTO MY_RAM;
    DEFAULT_ROM      INTO MY_ROM;
    SSTACK           INTO MY_STACK;

/* Place the section 'VectorTable' at the appropriated address. */
```

```

VectorTable      INTO Vector;

END

INIT ResetFunc

```

**NOTE**

The statement `Vector = READ_ONLY 0xFFFF8 TO 0xFFFF` defines the memory range for the vector table.

**NOTE**

The statement `VectorTable INTO Vector` specifies that the vector table should be loaded in the read only memory area `Vector`. This means, the constant `IRQ1Int` will be allocated at address `0xFFFF8`, the constant `IRQ0Int` will be allocated at address `0xFFFFA`, the constant `SWIInt` will be allocated at address `0xFFFFC`, and the constant `ResetInt` will be allocated at address `0xFFFFE`.

**NOTE**

The '+' after the object file name switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

### 14.3.3 Initializing the Vector Table in a source file using an absolute section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See the following listing for an example.

#### Listing: Initializing the Vector Table using an absolute section

```

XDEF ResetFunc
DataSec: SECTION

Data:    DS.W 5 ; Each interrupt increments an element of the table.

CodeSec: SECTION

```

## Initializing Vector table

```

; Implementation of the interrupt functions.

IRQ1Func:

    LDA    #0
    BRA    int

SWIFunc:

    LDA    #4
    BRA    int

ResetFunc:

    LDA    #8
    BRA    entry

DummyFunc:

    RTI

int:

    PSHH

    LDHX  #Data    ; Load address of symbol Data in X
                ; X <- address of the appropriate element in the tab

Offset:  TSTA

        BEQ    Offset3

Offset2:

    AIX    #1
    DECA

        BNE    Offset2

Offset3:

    INC    0, X    ; The table element is incremented

    PULH

    RTI

entry:

    LDHX  #0E00 ; Init Stack Pointer to $E00-$1=$DFF
    TXS

    CLRX

    CLRH

    CLI                ; Enables interrupts

loop:   BRA    loop

        ORG    $FFF8

; Definition of the vector table in an absolute section

```

```
; starting at address $FFF8.
IRQ1Int:  DC.W  IRQ1Func
IRQ0Int:  DC.W  DummyFunc
SWIInt:   DC.W  SWIFunc
ResetInt: DC.W  ResetFunc
```

The section should now be placed at the expected address. This is performed in the linker parameter file, as listed in the following listing:

### Listing: Example linker parameter file

```
LINK test.abs
NAMES

    test.o+

END

SECTIONS

    MY_ROM    = READ_ONLY  0x0800 TO 0x08FF;
    MY_RAM    = READ_WRITE 0x0B00 TO 0x0CFF;
    MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;

END

PLACEMENT

    DEFAULT_RAM      INTO MY_RAM;
    DEFAULT_ROM      INTO MY_ROM;
    SSTACK           INTO MY_STACK;

END

INIT ResetFunc
```

### NOTE

The '+' after the object file name switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

## 14.4 Splitting Application into Modules

Complex application or application involving several programmers can be split into several simple modules. In order to avoid any problem when merging the different modules, the following rules must be followed.

For each assembly source file, one include file must be created containing the definition of the symbols exported from this module. For the symbols referring to code label, a small description of the interface is required.

### 14.4.1 Example of an Assembly File (Test1.asm)

See the following listing for an example `Test1.asm` include file.

#### Listing: Separating Code into Modules - Test1.asm

```

XDEF AddSource
XDEF Source

DataSec: SECTION

Source: DS.W 1

CodeSec: SECTION

AddSource:

    RSP

    ADD Source

    STA Source

    RTS

```

### 14.4.2 Corresponding include file (Test1.inc)

See the following listing for an example `Test1.inc` include file.

#### Listing: Separating Code into Modules - Test1.inc

```

XREF AddSource
; The AddSource function adds the value stored in the variable
; Source to the contents of the A register. The result of the
; computation is stored in the Source variable.
;
; Input Parameter: The A register contains the value that should be

```



```

;          added to the Source variable.
; Output Parameter: Source contains the result of the addition.
          XREF Source
; The Source variable is a 1-byte variable.

```

### 14.4.3 Example of Assembly File (Test2.asm)

The following listing shows another assembly code file module for this project.

#### Listing: Separating Code into Modules-Test2.asm

```

          XDEF entry
          INCLUDE "Test1.inc"

CodeSec: SECTION
entry:   RSP

          LDA   #$7
          JSR   AddSource

          BRA   entry

```

The application's \*.prm file should list both object files building the application. When a section is present in the different object files, the object file sections are concatenated into a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the \*.prm file.

#### 14.4.3.1 Example of a PRM file (Test2.prm)

#### Listing: Separating assembly code into modules-Test2.prm

```

LINK test2.abs /* Name of the executable file generated. */
NAMES

    test1.o

    test2.o /*Name of the object files building the application. */

END

SECTIONS

    MY_ROM = READ_ONLY 0x2B00 TO 0x2BFF; /* READ_ONLY mem. */

    MY_RAM = READ_WRITE 0x2800 TO 0x28FF; /* READ_WRITE mem. */

END

```

## using Direct Addressing Mode to Access Symbols

PLACEMENT

```

/* variables are allocated in MY_RAM */
DataSec, DEFAULT_RAM INTO MY_RAM;
/* code and constants are allocated in MY_ROM */
CodeSec, ConstSec, DEFAULT_ROM INTO MY_ROM;

END

INIT entry /* Definition of the application entry point. */
VECTOR ADDRESS 0xFFFFE entry /* Definition of the reset vector. */

```

### NOTE

The `CodeSec` section is defined in both object files. In `test1.o`, the `CodeSec` section contains the symbol `AddSource`. In `test2.o`, the `CodeSec` section contains the `entry` symbol. According to the order in which the object files are listed in the `NAMES` block, the function `AddSource` is allocated first and the `entry` symbol is allocated next to it.

## 14.5 Using Direct Addressing Mode to Access Symbols

There are different ways for the Assembler to use the direct addressing mode on a symbol:

- [Using the direct addressing mode to access external symbols](#),
- [Using the direct addressing mode to access exported symbols](#),
- [Defining symbols in the direct page](#),
- [Using the force operator](#), or
- [Using SHORT sections](#).

### 14.5.1 Using the direct addressing mode to access external symbols

External symbols, which should be accessed using the direct addressing mode, must be declared using the `XREF.B` directive. Symbols which are imported using `XREF` are accessed using the extended addressing mode.

#### Listing: Using direct addressing to access external symbols

```

XREF.B ExternalDirLabel
XREF   ExternalExtLabel

...

LDA   ExternalDirLabel ; Direct addressing mode is used.

...

LDA   ExternalExtLabel ; Extended addressing mode is used.

```

## 14.5.2 Using the direct addressing mode to access exported symbols

Symbols, which are exported using the `XDEF.B` directive, will be accessed using the direct addressing mode. Symbols which are exported using `XDEF` are accessed using the extended addressing mode.

### Listing: Using direct addressing to access exported symbols

```

XDEF.B DirLabel
XDEF   ExtLabel

...

LDA   DirLabel ; Direct addressing mode is used.

...

LDA   ExtLabel ; Extended addressing mode is used.

```

## 14.5.3 Defining symbols in the direct page

Symbols that are defined in the predefined `BSC` section are always accessed using the direct-addressing mode, as listed in the following listing:

### Listing: Defining symbols in the direct page

```

...
        BSC
DirLabel: DS.B 3
dataSec: SECTION
ExtLabel: DS.B 5
...
codeSec: SECTION

```

```

...
        LDA    DirLabel ; Direct addressing mode is used.
...
        LDA    ExtLabel ; Extended addressing mode is used.

```

## 14.5.4 Using the force operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode ( *Listing: Using a force operator*).

The supported force operators are:

- < or .B to force direct addressing mode
- > or .W to force extended addressing mode.

### Listing: Using a force operator

```

...
dataSec: SECTION

label:   DS.B 5

...

codeSec: SECTION

...

        LDA    <label ; Direct addressing mode is used.
        LDA    label.B ; Direct addressing mode is used.
...

        LDA    >label ; Extended addressing mode is used.
        LDA    label.W ; Extended addressing mode is used.

```

## 14.5.5 Using SHORT sections

Symbols that are defined in a section defined with the `SHORT` qualifier are always accessed using the direct addressing mode, as listed in the following listing:

### Listing: Using SHORT sections

```

...
shortSec: SECTION SHORT

```

```
DirLabel: DS.B 3
dataSec: SECTION
ExtLabel: DS.B 5
...
codeSec: SECTION
...
        LDA    DirLabel ; Direct addressing mode is used.
...
        LDA    ExtLabel ; Extended addressing mode is used.
```



## Chapter 15 Appendices

This document has the following appendices:

- [Global Configuration File Entries](#)
- [Local Configuration File Entries](#)
- [MASM Compatibility](#)
- [MCUasm Compatibility](#)
- [Assembler Messages](#)

### 15.1 Appendices

This document has the following appendices:

- [Global Configuration File Entries](#)
- [Local Configuration File Entries](#)
- [MASM Compatibility](#)
- [MCUasm Compatibility](#)
- [Assembler Messages](#)





## Chapter 16

# Global Configuration File Entries

This appendix documents the sections and entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [\[Installation\] Section](#)
- [\[Options\] Section](#)
- [\[XXX\\_Assembler\] Section](#)
- [\[Editor\] Section](#)

### 16.1 [Installation] Section

This topic describes installation section.

#### 16.1.1 Path

##### Arguments

Last installation path.

##### Description

Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

##### Example

```
Path=C:\install
```

## 16.1.2 Group

### Arguments

Last installation program group.

### Description

Whenever a tool is installed, the installation script stores the installation program group created into this variable.

### Example

```
Group=Assembler
```

## 16.2 [Options] Section

This topic describes options section.

### 16.2.1 DefaultDir

#### Arguments

Default directory to be used.

#### Description

Specifies the current directory for all tools on a global level. See also [DEFAULTDIR: Default current directory](#) environment variable.

#### Example

```
DefaultDir=C:\install\project
```

## 16.3 [XXX\_Assembler] Section

This section documents the entries that can appear in an [XXX\_Assembler] section of the `mcutools.ini` file.

### NOTE

XXX is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC08 Assembler, the name of this section would be [HC08\_Assembler].

### 16.3.1 SaveOnExit

#### Arguments

1/0

#### Description

1 if the configuration should be stored when the Assembler is closed, 0 if it should not be stored. The Assembler does not ask to store a configuration in either cases.

### 16.3.2 SaveAppearance

#### Arguments

1/0

#### Description

1 if the visible topics should be stored when writing a project file, 0 if not. The command line, its history, the windows position and other topics belong to this entry.

This entry corresponds to the state of the *Appearance* check box in the Save Configuration dialog box.

### 16.3.3 SaveEditor

#### Arguments

1/0

#### Description

If the editor settings should be stored when writing a project file, 0 if not. The editor setting contain all information of the *Editor Configuration* dialog box. This entry corresponds to the state of the check box *Editor Configuration* in the [Save Configuration Dialog Box](#).

### 16.3.4 SaveOptions

#### Arguments

1/0

#### Description

1 if the options should be contained when writing a project file, 0 if not.

This entry corresponds to the state of the *Options* check box in the [Save Configuration Dialog Box](#).

### 16.3.5 RecentProject0, RecentProject1

#### Arguments

Names of the last and prior project files

#### Description

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

#### Example

```
SaveOnExit=1
```

```
SaveAppearance=1
```

```
SaveEditor=1
```

```
SaveOptions=1
```

```
RecentProject0=C:\myprj\project.ini
```

```
RecentProject1=C:\otherprj\project.ini
```

## 16.4 [Editor] Section

This topic describes editor section.

### 16.4.1 Editor\_Name

#### Arguments

The name of the global editor

#### Description

Specifies the name of the editor used as 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.

### 16.4.2 Editor\_Exe

example

## Arguments

The name of the executable file of the global editor (including path).

## Description

Specifies the filename which is started to edit a text file, when the global editor setting is active.

## Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

## 16.4.3 Editor\_Opts

### Arguments

The options to use with the global editor

### Description

Specifies options (arguments), which should be used when starting 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 the content of this entry.

### Saved

Only with *Editor Configuration* set in the **File > Configuration Save Configuration** dialog box.

### Example

```
[Editor]
```

```
editor_name=IDF
```

```
editor_exe=C:\Freescale\prog\idf.exe
```

```
editor_opts=%f -g%l,%c
```

## 16.5 Example

The following listing shows a typical `mcutools.ini` file.

### Listing: Typical `mcutools.ini` file layout

```
[Installation]
Path=c:\Freescale

Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%1,%c

[Options]
DefaultDir=c:\myprj

[HC08_Assembler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```





## Chapter 17

# Local Configuration File Entries

This appendix documents the sections and 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 contains these sections:

- [\[Editor\] Section](#)
- [\[XXX\\_Assembler\] Section](#)
- [Example](#)

### 17.1 [Editor] Section

This topic describes the editor section.

#### 17.1.1 Editor\_Name

##### Arguments

The name of the local editor

##### Description

Specifies the name of the editor used as local editor. This entry has only a description effect. Its content is not used to start the editor.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

##### Saved

Only with `Editor Configuration` set in the *File > Configuration > Save Configuration* dialog box.

## 17.1.2 Editor\_Exe

### Arguments

The name of the executable file of the local editor (including path).

### Description

Specifies the filename with is started to edit 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.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.1.3 Editor\_Opts

### Arguments

The options to use with the local editor

### Description

Specifies options (arguments), which should be used when starting the local editor. If this entry is not present or empty, `%f` is used. The command line to launch the editor is build by taking the `Editor_Exe` content, then appending a space followed by the content of this entry.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## Example

```
[Editor]

editor_name=IDF

editor_exe=C:\Freescale\prog\idf.exe

editor_opts=%f -g%1,%c
```

## 17.2 [XXX\_Assembler] Section

This section documents the entries that can appear in an [XXX\_Assembler] section of a `project.ini` file.

### NOTE

xxx is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC08 Assembler, the name of this section would be

```
[HC08_Assembler].
```

### 17.2.1 RecentCommandLineX, X= integer

#### Arguments

String with a command line history entry, e.g., `fibonacci.asm`

#### 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.

## 17.2.2 CurrentCommandLine

### Arguments

String with the command line, e.g., `fibonacci.asm -w1`

### Description

The currently visible command line content.

### Saved

Only with *Appearance* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.3 StatusBarEnabled

### Arguments

1/0

### Special

This entry is only considered at startup. Later load operations do not use it any more.

### Description

Current status bar state.

- 1: Status bar is visible
- 0: Status bar is hidden

### Saved

Only with *Appearance* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.4 ToolbarEnabled

### Arguments

1/0

### Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

### Description

Current toolbar state:

- 1: Toolbar is visible
- 0: Toolbar is hidden

### Saved

Only with *Appearance* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.5 WindowPos

### Arguments

10 integers, e.g., 0, 1, -1, -1, -1, -1, 390, 107, 1103, 643

### Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Changes of this entry do not show the "\*" in the title.

### Description

This numbers contain the position and the state of the window (maximized, etc.) and other flags.

### Saved

Only with *Appearance* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.6 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.

## Saved

Only with *Appearance* set in the **File > Configuration > Save Configuration** dialog box.

## Example

```
WindowFont=-16,500,0,Courier
```

## 17.2.7 TipFilePos

### Arguments

any integer, e.g., 236

### Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

### Saved

Always when saving a configuration file.

## 17.2.8 ShowTipOfDay

### Arguments

0/1

### Description

Should the *Tip of the Day* dialog box be shown at startup?

- 1: It should be shown
- 0: No, only when opened in the help menu

### Saved

Always when saving a configuration file.

## 17.2.9 Options

### Arguments

current option string, e.g.: `-w2`

### Description

The currently active option string. This entry can be very long.

### Saved

Only with *Options* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.10 EditorType

### Arguments

0/1/2/3/4

### 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`
- 4: CodeWarrior with COM. There are no additional entries.

For details, see also [Editor Setting Dialog Box](#).

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.11 EditorCommandLine

### Arguments

Command line, for UltraEdit-32: "`c:\Programs Files\IDM Software Solutions\UltraEdit-32\uedit32.exe %f -g%l,%c`"

### Description

Command line content to open a file. For details, see also [Editor Setting Dialog Box](#).

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.12 EditorDDEClientName

### Arguments

client command, e.g., "`[open(%f)]`"

### Description

Name of the client for DDE editor configuration. For details, see also [Editor Setting Dialog Box](#).

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.13 EditorDDETopicName

### Arguments

Topic name, e.g., `system`



## Description

Name of the topic for DDE editor configuration. For details, see also [Editor Setting Dialog Box](#).

## Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.2.14 EditorDDEServiceName

### Arguments

service name, e.g., `system`

### Description

Name of the service for DDE editor configuration. For details, see also Editor Setting dialog box.

### Saved

Only with *Editor Configuration* set in the **File > Configuration > Save Configuration** dialog box.

## 17.3 Example

The example in the following listing shows a typical layout of the configuration file (usually `project.ini`).

### Listing: Example of a project.ini file

```
[Editor]
Editor_Name=IDF

Editor_Exec=c:\Freescale\prog\idf.exe

Editor_Opts=%f -g%1,%c

[HC08_Assembler]

StatusBarEnabled=1

ToolBarEnabled=1
```

**example**

```
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.asm -w2
RecentCommandLine1=fibo.asm
CurrentCommandLine=fibo.asm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=c:\Freescale\prog\idf.exe %f -g%1,%c
```

## Chapter 18

# MASM Compatibility

The Macro Assembler has been extended to ensure compatibility with the MASM Assembler.

### 18.1 Comment Line

A line starting with a (\*) character is considered to be a comment line by the Assembler.

### 18.2 Constants (Integers)

For compatibility with the MASM Assembler, the following notations are also supported for integer constants:

- A decimal constant is defined by a sequence of decimal digits ( 0-9) followed by a `d` or `D` character.
- A hexadecimal constant is defined by a sequence of hexadecimal digits ( 0-9, a-f, A-F) followed by a `h` or `H` character.
- An octal constant is defined by a sequence of octal digits ( 0-7) followed by an `o`, `O`, `q`, or `Q` character.
- A binary constant is defined by a sequence of binary digits ( 0-1) followed by a `b` or `B` character.

#### Listing: Example

```
512d      ; decimal representation
512D      ; decimal representation

200h      ; hexadecimal representation
200H      ; hexadecimal representation
```

## Operators

```

1000o      ; octal representation
10000     ; octal representation
1000q     ; octal representation
1000Q     ; octal representation
1000000000b ; binary representation
1000000000B ; binary representation

```

## 18.3 Operators

For compatibility with the MASM Assembler, the notations listed in the following table are also supported for operators:

**Table 18-1. Operator notation for MASM compatibility**

Operator	Notation
Shift left	!<
Shift right	!>
Arithmetic AND	!.
Arithmetic OR	!+
Arithmetic XOR	!x, !X

### 18.3.1 Directives

The following table enumerates the directives that are supported by the Macro Assembler for compatibility with MASM:

**Table 18-2. Supported MASM directives**

Operator	Notation	Description
RMB	DS	Defines storage for a variable. Argument specifies the byte size.
RMD	DS 2*	Defines storage for a variable. Argument specifies the number of 2-byte blocks.
RMQ	DS 4*	Defines storage for a variable. Argument specifies the number of 4-byte blocks.
ELSEC	ELSE	Alternate of conditional block.
ENDC	ENDIF	End of conditional block.

*Table continues on the next page...*

**Table 18-2. Supported MASM directives (continued)**

Operator	Notation	Description
NOL	NOLIST	Specify that no subsequent instructions must be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (visible from outside)
PUBLIC	XDEF	Make a symbol public (visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.
XREFB	XREF.B	Import reference to an external symbol located on the direct page.
SWITCH		Allows switching to a previously defined section.
ASCT		Creates a predefined section named id ASCT.
BSCT		Creates a predefined section named id BSCT. Variables defined in this section are accessed using the direct addressing mode.
CSCT		Creates a predefined section named id CSCT.
DSCT		Creates a predefined section named id DSCT.
IDSCT		Creates a predefined section named id IDSCT.
IPSCT		Creates a predefined section named id IPSCT.
PSCT		Creates a predefined section named id PSCT.



## Chapter 19

# MCUasm Compatibility

The Macro Assembler has been extended to ensure compatibility with the MCUasm Assembler.

MCUasm compatibility mode can be activated, specifying the `-MCUasm` option.

This chapter covers the following topics:

- [Labels](#)
- [SET directive](#)
- [Obsolete directives](#)

### 19.1 Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even when they start on column 1.

When MCUasm compatibility mode is activated, following portion of code generate an error message, because the label `label` is not followed by a colon.

#### Listing: Example

```
label      DC.B 1
```

When MCUasm compatibility mode is not activated, the previous portion of code does not generate any error message.

### 19.2 SET directive

## Obsolete directives

When MCUasm compatibility mode is activated, relocatable expressions are also allowed in a SET directive.

When MCUasm compatibility mode is activated, the following portion of code does not generate any error messages:

### Listing: Example

```
label: SET *
```

When MCUasm compatibility mode is not activated, the previous portion of code generates an error message because the SET label can only refer to the absolute expressions.

## 19.3 Obsolete directives

The following table enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.

**Table 19-1. Obsolete directives**

Operator	Notation	Description
RMB	DS	Define storage for a variable
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (visible from the outside)
PUBLIC	XDEF	Make a symbol public (visible from the outside)
EXTERNAL	XREF	Import reference to an external symbol.



## Chapter 20

# Assembler Messages

This chapter describes the assembler messages.

### NOTE

Not all messages have been defined for this release. All descriptions will be available in an upcoming release.

## 20.1 Assembler Messages

Following is the list of the assembler messages.

### 20.1.1 A1: Unknown message occurred

[FATAL]

#### Description

The application tried to emit a message which was not defined. This is an internal error which should not occur. Please report any occurrences to your support.

#### Tips

Try to find out the and avoid the reason for the unknown message.

When you are generating an absolute file, your application should be encoded in a single source file, and should only contain absolute symbols. So in order to avoid this message, define all your sections as absolute sections and remove all XREF directives from your source file.

#### Example

```
                ORG $1000
Data1:          DS.W 1
                ORG $800
entry:
                NOP
                NOP
addData1:      DC.W Data1
```

## 20.1.2 A2: Message overflow, skipping <kind> messages

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

The application did show the number of messages of the specific kind as controlled with the options `\c -WmsgNi`, `\c -WmsgNw` and `\c -WmsgNe`. Further options of this kind are not displayed.

### Tips

Use the options `-WmsgNi`, `-WmsgNw` and `-WmsgNe` to change the number of messages.

## 20.1.3 A50: Input file '<file>' not found

[FATAL]

### Description

The Application was not able to find a file needed for processing.

### Tips

Check if the file really exists. Check if you are using a file name containing spaces (in this case you have to quote it).

### 20.1.4 A51: Cannot open statistic log file ``<file>`'

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

It was not possible to open a statistic output file, therefore no statistics are generated.

#### Note

Not all tools support statistic log files. Even if a tool does not support it, the message still exists, but is never issued in this case.

### 20.1.5 A52: Error in command line ``<cmd>`'

[FATAL]

#### Description

In case there is an error while processing the command line, this message is issued.

### 20.1.6 A53: Message `<Messageld>` is not used by this version. The mapping of this message is ignored.

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

The given message id was not recognized as known message. Usually this message is issued with the options `-WmsgS[D|I|W|E]<Num>` which should map a specific message to a different message kind.

#### Example

```
-WmsgSD123456789
```

#### Tips

There are various reasons why the tool would not recognize a certain message:

- make sure you are using the option with the right tool, say you don't disable linker messages in the compiler preferences
- The message may have existed for an previous version of the tool but was removed for example because a limitation does no longer exist.
- The message was added in a more recent version and the used old version did not support it yet.
- The message did never exist. Maybe a typo?

### 20.1.7 A54: Option <Option> .

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

This information is used to inform about special cases for options. One reason this message is used is for options which a previous version did support but this version does no longer support. The message itself contains a descriptive text how to handle this option now.

#### Tips

Check the manual for all the current option. Check the release notes about the background of this change.

### 20.1.8 A56: Option value overridden for option <OptionName>. Old value `<OldValue>', new value `<NewValue>'.

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

This message occurs when same option is specified more than once with same or different option values.

### 20.1.9 A64: Line Continuation occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

In any environment file, the character '\ ' at the end of a line is taken as line continuation. This line and the next one are handles as one line only. Because the path separation character of MS-DOS is also '\ ', paths are often incorrectly written ending with '\ '. Instead use a '.' after the last '\ ' to not finish a line with '\ ' unless you really want a line continuation.

## Example

Current Default.env:

```
...
LIBPATH=c:\Codewarrior\lib\
OBJPATH=c:\Codewarrior\work
...
```

Is taken identical as

```
...
LIBPATH=c:\Codewarrior\libOBJPATH=c:\Codewarrior\work
...
```

## Tips

To fix it, append a '.' behind the '\ '

```
...
LIBPATH=c:\Codewarrior\lib\
OBJPATH=c:\Codewarrior\work
...
```

**Note** Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as `64: Line Continuation occurred in <FileName>.`

## 20.1.10 A65: Environment macro expansion message '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

During a environment variable macro substitution an problem did occur. Possible causes are that the named macro did not exist or some length limitation was reached. Also recursive macros may cause this message.

### Example

Current variables:

```
...  
LIBPATH=${LIBPATH}  
...
```

### Tips

Check the definition of the environment variable.

## 20.1.11 A66: Search path <Name> does not exist

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

The tool did look for a file which was not found. During the failed search for the file, a non existing path was encountered.

### Tips

Check the spelling of your paths. Update the paths when moving a project. Use relative paths.

## 20.1.12 A1000: Conditional directive not closed

[ERROR]

## Description

One of the conditional blocks is not closed. A conditional block can be opened using one of the following directives: IF, IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE, IFC, IFNC, IFDEF, IFNDEF.

## Example

```
    IFEQ (defineConst)
const1: DC.B 1
const2: DC.B 2
```

## Tips

Close the conditional block with an ENDIF or ENDC directive.

## Example

```
    IFEQ (defineConst)
const1: DC.B 1
const2: DC.B 2
    ENDF
```

Be careful: A conditional block, which starts inside of a macro, must be closed within the same macro.

## Example

The following portion of code generates an error, because the conditional block "IFEQ" is opened within the macro "MyMacro" and is closed outside from the macro.

```
MyMacro: MACRO
    IFEQ (SaveRegs)
        DC.B 1
        DC.B 1
    ENDM
    DC.B 1
ENDIF
```

### 20.1.13 A1001: Conditional else not allowed here

[ERROR]

#### Description

A second ELSE directive is detected in a conditional block.

#### Example

```
    IFEQ (defineConst)
...
    ELSE
...
    ELSE
...
    ENDF
```

#### Tips

Remove the superfluous ELSE directive.

#### Example

```
    IFEQ (defineConst)
...
    ELSE
...
    ENDF
```

### 20.1.14 A1002: CASE, DEFAULT or ENDSW detected outside from a SWITCH block

[ERROR]

#### Description

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014



In Avocet compatibility mode, a CASE, DEFAULT or ENDSW directive was found without a previous SWITCH directive.

### Note

This message does only occur for assemblers supporting the Avocet compatibility mode.

### Example

```
xxx:    equ 0
;SWITCH xxx
    CASE 1
        DC.B 100
    CASE 2
        DC.B 200
    CASE 4
        DC.B 400
    DEFAULT
        FAIL 1
ENDSW
```

### Tips

Remove the semicolon in the example. Make sure that your assembler does support the Avocet compatibility mode and that this mode is switched on.

## 20.1.15 A1003: CASE or DEFAULT is missing

[ERROR]

### Description

In Avocet compatibility mode, after a SWITCH directive, an expression other than a CASE or DEFAULT entry was found.

### Note

This message does only occur for assemblers supporting the Avocet compatibility mode.

### Example

```

xxx:    equ 0

SWITCH xxx
    ; CASE 1
        DC.B 0

CASE 2
        DC.W 0

CASE 4
        DC.L 0

DEFAULT

FAIL 1

ENDSW

```

## Tips

Remove the semicolon in the example. Make sure that your assembler does support the Avocet compatibility mode and that this mode is switched on.

## 20.1.16 A1004: Macro nesting too deep. Possible recursion? Stop processing. (Set level with -MacroNest)

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

The macro expansion level was below the limit configured with the option -MacroNest.

### Example

In the following example, "\2" was used instead of the indented "/2". "\2" is taken by the assembler as second argument, which is not present and therefore it is replaced with the empty argument. Therefore this example leads to an endless macro recursion.

```

X_NOPs:  MACRO
    \@NofNops: EQU \1
        IF \@NofNops >= 1
            IF \@NofNops == 1
                NOP
            ELSE

```

```

        X_NOPs  \@NofNops\2
        X_NOPs  \@NofNops- (\@NofNops\2)
    ENDF
ENDF
ENDM
X_NOPs 17

```

## Tips

Use the option `-MacroNest` to configure the macro expansion level. In the above example, use `"/2"` to get the correct macro:

```

X_NOPs:  MACRO
    \@NofNops: EQU \1
        IF \@NofNops >= 1
            IF \@NofNops == 1
                NOP
            ELSE
                X_NOPs  \@NofNops/2
                X_NOPs  \@NofNops- (\@NofNops/2)
            ENDIF
        ENDIF
    ENDF
ENDM
X_NOPs 17

```

## See also

Option `-MacroNest`

### 20.1.17 A1051: Zero Division in expression

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

A zero division is detected in an expression.

#### Example

## Assembler Messages

```
label: EQU 0

label2: EQU $5000

        DC (label2/label)
```

### Tips

Modify the expression or specify it in a conditional assembly block.

### Example

```
label: EQU 0

label2: EQU $5000

        IFNE (label)
            DC (label2/label)
        ELSE
            DC label2
        ENDF
```

## 20.1.18 A1052: Right parenthesis expected

[ERROR]

### Description

A right parenthesis is missing in an assembly expression.

### Example

```
variable: DS.W 1

label1: EQU (2*4+6

label3: EQU LOW(variable

label4: EQU HIGH(variable
```

### Tips

Insert the right parenthesis at the correct position.

### Example

```
variable: DS.W 1  
label1: EQU (2*4+6)  
label3: EQU LOW(variable)  
label4: EQU HIGH(variable)
```

### 20.1.19 A1053: Left parenthesis expected

[ERROR]

#### Description

A left parenthesis is missing in an assembly expression.

#### Example

```
variable: ds.w 1  
label1: EQU LOW variable)  
label2: EQU HIGH variable)
```

#### Tips

Insert the left parenthesis at the correct position.

#### Example

```
label1: EQU LOW(variable)  
label2: EQU HIGH(variable)
```

### 20.1.20 A1054: References on non-absolute objects are not allowed when options -FA1 or -FA2 are enabled

[ERROR]

#### Description

A reference to a relocatable object has been detected during generation of an absolute file by the assembler.

## Example

```

        XREF extData

DataSec: SECTION

data1:  DS.W 1

        ORG $800

entry:

        DC.W extData

        DC.W data1+2

```

## Tips

When you are generating an absolute file, your application should be encoded in a single source file, and should only contain any relocatable symbol. So in order to avoid this message, define all your section as absolute section and remove all XREF directives from your source file.

## Example

```

        ORG $B00

data1:  DS.W 1

        ORG $800

entry:

        DC.W data1+2

```

## 20.1.21 A1055: Error in expression

[ERROR]

### Description

An error has been discovered in an expression while parsing it.

### Example

```

CodeSec2: SECTION

Entry2:

        LD A, # $08

```

```
label:   JRA    (Entry2 + 1
```

## Example

```
CodeSec2: SECTION
Entry2:
          LDA   #$08
label:   JMP   (Entry2 + 1
```

## Example

```
CodeSec2: SECTION
Entry2:
          LDAA  #$08
label:   JMP   (Entry2 + 1
```

## Example

```
CodeSec2: SECTION
Entry2:
          MOVE  #$08,D0
label:   JMP   (Entry2 +
```

## Tips

Correct the expression.

### 20.1.22 A1056: Error at end of expression

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

An error has been detected by the assembler at the end of the read expression.

#### Example

```
char: SET 1 this is a comment
```

## Tips

Remove the not correct symbol at the end of line or insert a comment start ";".

## Example

```
char: SET 1 ;this is a comment
```

### 20.1.23 A1057: Cutting constant because of overflow

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

A constant was truncated because of an overflow. Only the lower bits were used to generate the output.

#### Example

```
DC $123456789
```

## Tips

Only use 32 bit constants. Use several DC's to produce larger values.

### 20.1.24 A1058: Illegal floating point operation

[DISABLE, INFORMATION, WARNING, ERROR]

#### Description

An illegal floating point operation other than unary minus or unary plus has been detected.

### 20.1.25 A1059: != is taken as EQUAL

[DISABLE, INFORMATION, WARNING, ERROR]



## Description

The != operator is taken as equal. This behavior is different from the C language or the usual assembler behavior. The behavior is caused by the Option -Compat. Disable the message, if you are aware of the different semantic

## See also

Option -Compat

### 20.1.26 A1060: Implicit comment start

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

With the alternate comment syntax of the option Option -Compat=C, this message is issued if the ignored part does not start with a star ("\*") or with a semicolon (";").

## See also

Option -Compat

### 20.1.27 A1061: Floating Point format is not supported for this case

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

The floating point value is not supported at this place.

### 20.1.28 A1062: Floating Point number expected

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

The assembler did expect a floating point value, but he found an expression of a different type.

## Note

Not all assemblers do support floating point constants. Assemblers not supporting floating point do not issue this message.

## Example

```
; The example only works with assemblers supporting  
floating point with a dc.f directive label
```

```
dc.f label
```

## 20.1.29 A1101: Illegal label: label is reserved

[ERROR]

### Description

A reserved identifier is used as label. Reserved identifiers are the mnemonics associated with target processor registers and some additional Reserved Symbols.

### Example

```
X: SET 3
```

### Tips

Modify the name of the label to a identifier which is not reserved.

### Example

```
_X: SET 3
```

### See also

Reserved Symbols

## 20.1.30 A1103: Illegal redefinition of label

[ERROR]

## Description

The label specified in front of a comment or an assembly instruction or directive, is detected twice in a source file.

## Example

```
DataSec1: SECTION
label1: DS.W 2
label2: DS.L 2
...
label2: DS.W 3
```

## Tips

Modify the label names, in order to have unique label identification in each assembly file.

## Example

```
DataSec1: SECTION
label1: DS.W 2
label2: DS.L 2
...
label3: DS.W 3
```

### 20.1.31 A1104: Undeclared user defined symbol: <Symbol>

[ERROR]

## Description

The label <symbolName> is referenced in the assembly file, but it is never defined.

## Example

```
data: SECTION
count:
    DC.W counter
```

## Tips

The label <symbolName> must be either defined in the current assembly file or specified as an external label.

### Example

```

        XREF counter
data:   SECTION
count:
        DC.W counter

```

## 20.1.32 A1105: Closing brace expected

[ERROR]

### Description

The error occurs for missing closing brace of syntax {label}. Assembler replaces the label within { } with its value.

## 20.1.33 A1106: Undeclared user defined symbol: <Symbol>. Symbol name is expected as macro argument

[ERROR]

### Description

This message occurs in macro for undefined symbols. The symbol is being evaluated in braces({}).

### Example:

```

clear: MACRO
label{\1} EQU 1
        ENDM
; code section
MyCode: SECTION
main:

```

```
clear tmp
```

tmp symbol is undefined and the error is thrown.

### 20.1.34 A1201: Label <Label> referenced in directive ABSENTRY is not absolute

[ERROR]

#### Description

The label specified in the directive ABSENTRY is an EQU label or is located in a data section. The label specified in ABSENTRY must be a valid label defined in a code section.

#### Example

```
ABSENTRY const
const: EQU    $1000
      ORG    const
      DC.B   1
      DC.B   2
```

#### Tips

Specify a label defined in a code section in ABSENTRY or remove the directive ABSENTRY.

#### Example

```
ABSENTRY entry
const: EQU    $1000
      ORG    const
entry: DC.B   1
      DC.B   2
```

### 20.1.35 A1202: ELF output: <details>

[ERROR]

#### Description

Error in ELF. <details> specifies the cause of the error. Possible causes are:

- Cannot open <File>
- Currently no file open
- Request is not valid
- Internal

### 20.1.36 A1203:

[WARNING]

#### Description

This message occurs if incorrect debug information is generated.

### 20.1.37 A1251: Cannot open object file: Object file name too long

[ERROR]

#### Description

The object file is derived from the source file name by changing the extension to ".o". If the source file name is extremely long, then this may fail.

#### Tips

Use shorter filenames.

### 20.1.38 A1252: The exported label <name> is using an ELF extension

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

This message is only issued when using the ELF object file format. It can be ignored when using the linker, however, foreign linker may not know this extension and therefore the linking might fail. The exported label `\<name>` is using an ELF extension for exported labels, which are defined as imported label plus offset. This situation cannot be expressed in a standard ELF symbol table, so the assembler is generating a symbol with type `STT_LOPROC`. This message is disabled by default, so it does not occur unless it is explicitly enabled. When setting this message to an error, code containing such cases cannot be assembled.

## Example

```
XREF ImportedLabel  
  
ExportedLabel: EQU ImportedLabel + 1  
  
XDEF ExportedLabel
```

## Tips

Set this message to an error when you plan to use a foreign linker. Adapt the source code so that this case does not occur.

### 20.1.39 A1253: Limitation: code size > <SizeLimit> bytes

[ERROR]

#### Description

The assembler is running in demo mode and the code size limitation was reached. Therefore the assembly process is stopped.

#### Tips

Make sure the license is correctly installed. Check the about box about the current license state.

### 20.1.40 A1301: Structured type redefinition: <TypeName>

[ERROR]

## Description

The same name has been associated with two different structured types. <TypeName> is the name of the structured type, which is defined twice.

## Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

## Example

```
myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT
          XREF myData:myType
myType:  STRUCT
field3:   DS.B 1
field4:   DS.B 3
          ENDSTRUCT
```

## Tips

Change the name of one of the structured type.

## Example

```
myType1: STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT
          XREF myData:myType1
myType2: STRUCT
field3:   DS.B 1
field4:   DS.B 3
          ENDSTRUCT
```



## 20.1.41 A1302: Type <TypeName> is previously defined as label

[ERROR]

### Description

The identifier used to identify a structured type was previously used as a label. <TypeName> is the name of the structured type, which is already used as label name.

### Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

### Example

```
myType: DS.W 3
...
myType: STRUCT
field1: DS.W 1
field2: DS.W 1
        ENDSTRUCT
```

### Tips

Change the name of one of the structured type or of the label.

### Example

```
myVar: DS.W 3
...
myType: STRUCT
field1: DS.W 1
field2: DS.W 1
        ENDSTRUCT
```

## 20.1.42 A1303: No type defined

[ERROR]

## Description

A directive only allowed inside of a structured type definition was found without a leading STRUCT or UNION.

## Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

## Example

```
field1:    DS.W 1
field2:    DS.W 1
          ENDSTRUCT
```

## Tips

Check the STRUCT directive at the start.

## Example

```
myType:   STRUCT
field1:    DS.W 1
field2:    DS.W 1
          ENDSTRUCT
```

### 20.1.43 A1304: Field <FieldName> is not declared in specified type

[ERROR]

## Description

The field specified is not part of the structured type associated with the variable addressed. <FieldName> is the name of the field addressed in the variable.

## Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

## Example

```
myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT
          XREF myData:myType
          XDEF entry
CodeSec:  SECTION
entry:
          NOP
          NOP
          LD A, myData:field33
```

## Example

```
myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT
          XREF myData:myType
          XDEF entry
CodeSec:  SECTION
entry:
          NOP
          NOP
          LDX myData:field33
```

## Example

```
myType:  STRUCT
field1:  DS.B 1
field2:  DS.B 1
          ENDSTRUCT
          XREF myData:myType
const:   SECTION
          DC.W myData:field1
```

```
DC.W myData:field3 ; no field3
```

## Example

```
myType: STRUCT
field1: DS.B 1
field2: DS.B 1
ENDSTRUCT
XREF myData:myType
const: SECTION
LRW R4, [myDataField33]
LD.B R4, (R4,0)
BR *
ALIGN 4
myDataField33: DC.W myData:field33
```

## Tips

Change the name of the field to an existing field or define the field in the structured type.

## Example

```
myType: STRUCT
field1: DS.W 1
field2: DS.W 1
ENDSTRUCT
XREF myData:myType
XDEF entry
CodeSec: SECTION
entry:
NOP
NOP
LD A, myData:field2
```

## Example

```
myType: STRUCT
field1: DS.W 1
```

```
field2:    DS.W 1
           ENDSTRUCT
           XREF myData:myType
           XDEF entry
CodeSec: SECTION
entry:
           NOP
           NOP
           LDX myData:field2
```

## Example

```
myType: STRUCT
field1:  DS.B  1
field2:  DS.B  1
           ENDSTRUCT
           XREF  myData:myType
CodeSec: SECTION
           LRW R4, [myDataField2]
           LD.B  R4, (R4, 0)
           BR   *
           ALIGN 4
myDataField2: DC.W myData:field2
```

### 20.1.44 A1305: Type name expected

[ERROR]

#### Description

The symbol specified after a TYPE directive is not a previous defined structured type.

#### Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

#### Example

## Assembler Messages

```

myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT

DataSec: SECTION
myData:  TYPE yType
          XDEF entry

CodeSec: SECTION
entry:
          NOP
          NOP
          LD A, myData:field2

```

## Example

```

myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT

DataSec: SECTION
myData:  TYPE yType
          XDEF entry

CodeSec: SECTION
entry:
          NOP
          NOP
          LDX myData:field2

```

## Example

```

myType:  STRUCT
field1:   DS.B 1
field2:   DS.B 1
          ENDSTRUCT

data:    SECTION
myData:  TYPE yType ; no type "yType"

```

## Example

```
myType:  STRUCT
field1:   DS.W 1
field2:   DS.W 1
          ENDSTRUCT

DataSec: SECTION
myData:  TYPE yType
          XDEF entry

CodeSec: SECTION
entry:
          LRW R4, [myDataField2]
          LD.B   R4, (R4,0)
          BR    *
          ALIGN 4

myDataField2: DC.W myData:field2
```

## Tips

Change the name of the type for a valid type name.

## Example

```
myType: STRUCT
field1:  DS.W 1
field2:  DS.W 1
          ENDSTRUCT

DataSec: SECTION
myData:  TYPE myType
          XDEF entry

CodeSec: SECTION
entry:
          NOP
          NOP
          LD A, myData:field2
```

## Example

## Assembler Messages

```

myType:  STRUCT
field1:  DS.W 1
field2:  DS.W 1
        ENDSTRUCT
DataSec: SECTION
myData:  TYPE myType
        XDEF entry
CodeSec: SECTION
entry:
        NOP
        NOP
        LDX myData:field2

```

## Example

```

myType:  STRUCT
field1:  DS.W 1
field2:  DS.W 1
        ENDSTRUCT
DataSec: SECTION
myData:  TYPE myType
        XDEF entry
CodeSec: SECTION
entry:
        LRW R4, [myDataField2]
        LD.B   R4, (R4,0)
        BR    *
        ALIGN 4
myDataField2: DC.W myData:field2

```

## 20.1.45 A1401: Value out of range -128..127

[ERROR]



## Description

The offset between the current PC and the label specified as PC relative address is not in the range of a signed byte (smaller than -128 or bigger than 127). An 8 bit signed PC relative offset is expected in following instructions:

## Note

Not all assemblers do have instructions with 8 bit PC relative addressing mode.

Such assemblers will not issue this message at all.

Third operand in following instructions: BRCLR, BRSET

## Example

```
dataSec: SECTION
    var1:   DS.W 1
    var2:   DS.W 2
codeSec: SECTION
    LDA    var1
    CMP    9
    BNE    label
dummyB1: DCB.B 200, $A7
label    STA var2
```

## Tips

If you have used one of the branch instructions, use the JMP instruction as in the example below.

## Example

```
dataSec: SECTION
    var1:   DS.W 1
    var2:   DS.W 2
codeSec: SECTION
    LDA    var1
    CMP    9
    BEQ    continue    ; do not take branch
    JMP    label
continue:
```

## Assembler Messages

```

        NOP
        NOP
dummyB1: DCB.B 200, $A7
label   STA var2

```

### 20.1.46 A1402: Value out of range -32768..32767

[ERROR]

#### Description

The offset between the current PC and the label specified as PC relative address is not in the range of a signed word (smaller than -32768 or bigger than 32767).

#### Note

Not all assemblers do have instructions with 16 bit PC relative addressing mode. Such assemblers will not issue this message at all.

### 20.1.47 A1405: PAGE with initialized RAM not supported

[ERROR]

#### Description

The Macro Assembler does not support the use of the HIGH operator with initialized RAM in the HIWARE format. In the ELF format, it is allowed and this message is not used.

#### Note

Not all assemblers do support the PAGE operator.

#### Example

```

cstSec: SECTION
pgEntry DC.B PAGE(entry)
adrEntry: DC.W entry
codeSec: SECTION
entry:

```

```
    NOP
```

```
    NOP
```

## Tips

You can load the whole address from the entry label using a DC.L directive. The only draw back is that you have allocated 4 byte to store the address instead of 3 bytes.

## Example

```
cstSec: SECTION
adrEntry: DC.L entry
codeSec: SECTION
entry:
    NOP
    NOP
```

## 20.1.48 A1406: HIGH with initialized RAM not supported

[ERROR]

### Description

The Macro Assembler does not support the use of the HIGH operator with initialized RAM in the HIWARE format. In the ELF format, it is allowed and this message is not used.

### Note

Not all assemblers do support the HIGH operator.

### Example

```
MyData: SECTION
table: DS.W 1
       DC.B high(table)
```

## 20.1.49 A1407: LOW with initialized RAM not supported

[ERROR]

### Description

The Macro Assembler does not support the use of the LOW operator with initialized RAM in the HIWARE format. In the ELF format, it is allowed and this message is not used.

### Note

Not all assemblers do support the LOW operator.

### Example

```
MyData: SECTION
table: DS.W 1
      DC.B low(table)
```

## 20.1.50 A1408: Out of memory, Code size too large

[ERROR]

### Description

The assembler runs out of memory because of a very large section.

### Note

This assembler version does no longer have the 32k size limitation of previous versions.

## 20.1.51 A1410: EQU or SET labels are not allowed in a PC relative addressing mode

[ERROR]

### Description

An absolute EQU or SET label has been detected in an indexed PC relative addressing mode. This is not legal in a relocatable expression.

### Note

Not all assemblers do have special PC Relative addressing modes. Such assemblers will not issue this message at all.

### Example

```
label:      EQU $FF30
dataSec:    SECTION
data:       DS.W 1
codeSec1:   SECTION
entry:
            LDD label, PCR
            STD data
```

### Tips

Make the section an absolute section.

### Example

```
label:      EQU $FF30
dataSec:    SECTION
data:       DS.W 1
            ORG $C000
entry:
            LDD label, PCR
            STD data
```

## 20.1.52 A1411: PC Relative addressing mode is not supported to constants

[ERROR]

### Description

An absolute expression has been detected in an indexed PC relative addressing mode. This is not legal in a relocatable expression. Not all assemblers do have special PC Relative addressing modes. Such assemblers will not issue this message at all.

### Example

```
dataSec: SECTION
data: DS.W 1
codeSec1: SECTION
entry:
    LDD $FF35, PCR
    STD data
```

### Tips

Make the section an absolute section.

### Example

```
dataSec: SECTION
data: DS.W 1
    ORG $C000
entry:
    LDD $FF35, PCR
    STD data
```

## 20.1.53 A1412: Relocatable object '<Symbol>' not allowed if generating absolute file

[ERROR]

### Description

No relocatable objects are allowed if the user requests the generation of an absolute file. This message occurs primarily for objects in the default (relocatable) section.

### Example

```
ABSENTRY main
```

```
main: DC.B 1
      DC.B 2
```

## Tips

Place all objects into absolute sections.

## Example

```
ABSENTRY main
ORG $1000
main: DC.B 1
      DC.B 2
```

### 20.1.54 A1413: Value out of relative range

[DISABLED, INFORMATION, WARNING, **ERROR** ]

#### Description

Some value did not fit into the operand field of an instruction. This message can be disabled if the value should be just truncated.

#### Tips

Check if you can place the code and the referenced object closer together. Try to generate a smaller displacement. If this is not possible, consider using another instruction or addressing mode.

### 20.1.55 A1414: Cannot set fixup

[ERROR]

#### Description

The assembler cannot set a fixup because the referenced object is just a constant rather than an object. One case when the assembler must generate a fixup are PCR relative accesses in relocatable code. Then the assembler does need an object which refers to the accessed address.

## Tips

Check why the assembler has to set a fixup instead of just using a constant.

### 20.1.56 A1415: Cutting fixup overflow

[DISABLED, INFORMATION, WARNING, ERROR]

#### Description

A constant value does not fit into a field and is therefore truncated.

#### Example

```
        DC.B Label+1
Label: EQU $ff
        DC.B Label+1
```

## Tips

Use a larger field, if necessary.

```
        DC.W Label+1
Label: EQU $ff
        DC.W Label+1
```

### 20.1.57 A1416: Absolute section starting at <Address> size <Size> overlaps with absolute section starting at <Address>

[DISABLED, INFORMATION, WARNING, ERROR]

#### Description

Two absolute sections are overlapping each other.

#### Example



```
ORG $1000

DC.B 0,1,2,3

; address $1004

DA: SECTION

DC.B 1

ORG $1001

DC.B 0,1,2,3

; address $1005
```

## Tips

- Use non overlapping areas, whenever possible.
- Use relocatable sections if you want to split up a memory area into several modules.
- Calculate the start address of the second with the end address of the first, if they are in the same assembly unit.

## Example

```
ORG $1000

DC.B 0,1,2,3

; address $1004

SectEnd: EQU *

DA: SECTION

DC.B 1

ORG SectEnd

DC.B 0,1,2,3

; address $1008
```

### 20.1.58 A1417: Value out of possible range: Value<value> not in range [<LowValue>..<HighValue>]

[DISABLED, INFORMATION, WARNING, **ERROR** ]

#### Description

A constant value does not fit into a field. This message is used to stop the assembly for some fixup overflow cases.

## Tips

Usually this message is used for branch distances, if so, try to use a branch with a larger range.

### 20.1.59 A1418: Negative offset is not propagated into the page in the fixup

[DISABLED, INFORMATION, WARNING, ERROR]

#### Description

The assembler did ignore a negative page offset.

#### Example

```
labelStart:  
    DC.W labelStart - 1  
    DC.B page(labelStart - 1);
```

## Tips

Usually negative offsets do intend to access the same page.

### 20.1.60 A1419: Constant fixup failure <Description>

[DISABLED, INFORMATION, WARNING, ERROR]

#### Description

The assembler was not able to compute a specific fixup at assemble time. This message is issued if certain properties of a specific fixup are not met. For example if a fixup does only handle even addresses, but the address passed in turned out to be odd.

## Tips

Check if you did use the right fixup type. Check if the actual instruction is legal with this fixup type and with the passed in value.

### 20.1.61 A1502: Reserved identifiers are not allowed as instruction or directive

[ERROR]

#### Description

The identifier detected in an assembly line instruction part is a Reserved Symbol.

#### See also

Reserved Symbols

### 20.1.62 A1503: Error in option -D: `**<Description>**'

[DISABLED, INFORMATION, WARNING, **ERROR** ]

#### Description

An option -D was used with illegal content. The format for -D is "-D" name ["="value]. The name must be a legal for a label. The value must be a number. There must be a number after an equal ("=").

#### Example

```
Not a legal label name:
```

```
-D1
```

```
After a =, there must be a value:
```

```
-DLabelName=
```

```
Unexpected text at the end:
```

```
-D"LabelName1=1 1"
```

#### See also

Option -D

### 20.1.63 A1601: Label must be terminated with a <:;>

[ERROR]

## Description

This message is issued only when labels must be terminated with a colon. For some targets, this is not required. Then this message is not issued. This message is only generated when the MCUasm compatibility is switched on. In this case, all labels must be terminated with a colon (:) character.

### 20.1.64 A1602: Invalid character at end of label (<LabelName>): semicolon or space expected

[ERROR]

## Description

The specified label is terminated by an invalid character. The following characters are allowed in a label:

- All alphabetical characters ('a'.. 'z', 'A', 'Z').
- All numerical characters (`0' .. `9').
- `.' and `\_'

<LabelName> is the name of the wrong label detected (including the invalid character).

## Example

```
Data1#    DS.B 1
Data2#6   DS.B 1
```

## Tips

Remove the invalid character or replace it by a '\_'.

## Example

```
Data1     DS.B 1
Data2_6   DS.B 1
```

### 20.1.65 A1603: Directive, instruction or macro name expected: <Symbol> detected

[ERROR]

## Description

The symbol detected in the operation field is not a valid directive, instruction or macro name. <SymbolName> is the name of the invalid string detected in the operation field.

## Example

```
label:    XXX 3
label2:   label
```

## Tips

Replace the specified symbolName by a valid instruction, directive or macro name.

## 20.1.66 A1604: Invalid character detected at the beginning of the line: <Character>

[ERROR]

## Description

The character detected on column 1 is not valid. For the macro assembler everything starting on column 1 is supposed to be a label. The following characters are allowed at the beginning of a label:

- All alphabetical characters ('a'.. 'z', 'A', 'Z').
- `.' and `\_'

<Character> is the character detected on column 1

## Example

```
@label:   DS.B 1
4label2:  DS.B 2
```

## Tips

Replace the specified character by a '.' or a '\_'.

## Example

```
_label:    DS.B 1  
.label2:  DS.B 2
```

## 20.1.67 A1605: Invalid label name: <LabelName>

[ERROR]

### Description

The character detected at the beginning of a label is not valid. The following characters are allowed at the beginning of a label:

- All alphabetical characters ('a'.. 'z', 'A', 'Z').
- `.'and `\_'.

<LabelName> is the label name detected.

### Example

```
#label:    DS.B 1
```

### Tips

Replace the specified character by a `.'or a `\_'

### Example

```
_label:    DS.B 1
```

## 20.1.68 A2301: Label is missing

[ERROR]

### Description

A label name is missing on the front of an assembly directive requiring a label. These directives are:

- SECTION

- EQU
- SET

### Example

```
SECTION 4
...
EQU $67
...
SET $77
```

### Tips

Insert a label in front of the directive.

### Example

```
codeSec: SECTION 4
...
myConst: EQU $67
...
mySetV: SET $77
```

## 20.1.69 A2302: Macro name is missing

[ERROR]

### Description

A label name is missing on the front of a MACRO directive.

### Example

```
MyData: SECTION
Data1: DS.B 1
      MACRO
      DC.B \1
      ENDM
MyCode: SECTION
```

Entry:

## Tips

Insert a label in front of the MACRO directive.

## Example

```
MyData: SECTION
Data1: DS.B 1
allocChar: MACRO
          DC.B \1
          ENDM
MyCode: SECTION
Entry:
```

## 20.1.70 A2303: ENDM is illegal

[ERROR]

## Description

A ENDM directive is detected outside of a macro.

## Example

```
MyData: SECTION
Data1: DS.B 1
allocChar: MACRO
          DC.B \1
          ENDM
MyCode: SECTION
Entry:
          ENDM
```

## Tips

Remove the superfluous ENDM directive.



## Example

```
MyData: SECTION
Data1: DS.B 1
allocChar: MACRO
    DC.B \1
ENDM
MyCode: SECTION
Entry:
```

### 20.1.71 A2304: Macro definition within definition

[ERROR]

#### Description

A macro definition is detected inside of another macro definition. The macro assembler does not support this.

#### Example

```
allocChar: MACRO
allocWord: MACRO
    DC.W \1
ENDM
    DC.B \1
ENDM
```

#### Tips

Define the second macro outside from the first one.

#### Example

```
allocChar: MACRO
    DC.B \1
ENDM
allocWord: MACRO
```

```
DC.W \1
ENDM
```

## 20.1.72 A2305: Illegal redefinition of instruction or directive name

[ERROR]

### Description

An assembly directive or a mnemonic has been used as macro name. This is not allowed to avoid any ambiguity when the symbol name is encountered afterward. The macro assembler cannot detect if the symbol refers to the macro or the instruction.

### Example

```
DC: MACRO
DC.B \1
ENDM
```

### Tips

Change the name of the macro to an unused identifier.

### Example

```
allocChar: MACRO
DC.B \1
ENDM
```

## 20.1.73 A2306: Macro not closed at end of source

[ERROR]

### Description

An ENDM directive is missing at the end of a macro. The end of the input file is detected before the end of the macro.

## Example

```
allocChar: MACRO
    DC.B \1

myData: SECTION SHORT
char1: DS.B 1
char2: DS.B 1
myConst: SECTION SHORT
init1: DC.B $33
init2: DC.B $43
...
```

## Tips

Insert the missing ENDM directive at the end of the macro.

## Example

```
allocChar: MACRO
    DC.B \1
    ENDM

myData: SECTION SHORT
char1: DS.B 1
char2: DS.B 1
myConst: SECTION SHORT
init1: DC.B $33
init2: DC.B $43
...
```

### 20.1.74 A2307: Macro redefinition

[DISABLE, INFORMATION, **WARNING** , ERROR]

#### Description

The input file contains the definition of two macros, which have the same name.

#### Example

```
alloc: MACRO
    DC.B \1
ENDM

alloc: MACRO
    DC.W \1
ENDM
```

## Tips

Change the name of one of the macros to generate unique identifiers.

## Example

```
allocChar: MACRO
    DC.B \1
ENDM

allocWord: MACRO
    DC.W \1
ENDM
```

## 20.1.75 A2308: File name expected

[ERROR]

### Description

A file name is expected in an INCLUDE directive.

### Example

```
INCLUDE 1234
```

## Tips

Specify a file name after the INCLUDE directive.

## Example

```
INCLUDE "1234" ; file is named "1234"
```

### 20.1.76 A2309: File not found

[ERROR]

#### Description

The assembler cannot find the file, which name is specified in the include directive.

#### Tips

- If the file exist, check if the directory where it is located is specified in the GENPATH environment variable.
- First check if your project directory is correct. A file "default.env" should be located there, where the environment variables are stored.
- The macro assembler looks for the included files in the working directory and then in the directory enumerated in the \c GENPATH environment variable.
- If the file do not exist, create it or remove the include directive.

### 20.1.77 A2310: Size specification expected

[ERROR]

#### Description

An invalid size specification character is detected in a DCB, DC, DS, FCC, FCB, FDB, FQB, RMB, XDEF or XREF directive. For XDEF and XREF directives, valid size specification characters are:

- .B: for symbols located in a section where direct addressing mode can be used.
- .W: for symbols located in a section where extended addressing mode must be used.

For DCB, DC, DS, FCC, FCB, FDB, FQB and RMB directives, valid size specification characters are:

- .B: for Byte variables.
- .W: for Word variables.
- .L: for Long variables.

## Example

```
DataSec: SECTION
label1: DS.Q 2
ConstSec: SECTION
label2: DC.I 3, 4, 66
```

## Tips

Change the size specification character to a valid one.

## Example

```
DataSec: SECTION
label1: DS.W 2
ConstSec: SECTION
label2: DC.W 3, 4, 66
```

## 20.1.78 A2311: Symbol name expected

[ERROR]

### Description

A symbol name is missing after a XDEF, XREF, IFDEF or IFNDEF directive.

### Example

```
XDEF $5645
XREF ; This is a comment
CodeSec: SECTION
IFDEF $5634
ENDIF
```

## Tips

Insert a symbol name at the requested position.

## Example

```
XDEF exportedSymbol
XREF importedSymbol; This is a comment
CodeSec: SECTION
exportedSymbol:
    IFDEF changeBank
    ENDIF
```

## 20.1.79 A2312: String expected

[ERROR]

### Description

A character string is expected at the end of a FCC, IFC or IFNC directive.

### Example

```
one:    MACRO
        IFC \1, ""
        DS.B 1
        ELSE
        DC.B \1
        ENDIF
        ENDM
one    $42
```

### Tips

Insert a character string at the requested position.

### Example

```
one:    MACRO
        IFC "\1", ""
        DS.B 1
        ELSE
        DC.B \1
        ENDIF
```

```
ENDM  
one $42
```

## 20.1.80 A2313: Nesting of include files exceeds 50

[ERROR]

### Description

The maximum number of nested include files has been exceeded. The Macro Assembler supports up to 50 nested include files.

### Tips

Reduce the number of nested include file to 50.

## 20.1.81 A2314: Expression must be absolute

[ERROR]

### Description

An absolute expression is expected at the specified position. Assembler directives expecting an absolute value are: OFFSET, ORG, ALIGN, SET, BASE, DS, LLEN, PLEN, SPC, TABS, IF, IFEQ, IFNE, IFLE, IFLT, IFGE, IFGT. The first operand in a DCB directive must be absolute:

### Example

```
DataSec: SECTION  
label1: DS.W 1  
label2: DS.W 2  
label3: EQU 8  
codeSec: SECTION  
        BASE label1  
        ALIGN label2
```

### Tips



Specify an absolute expression at the specified position.

### Example

```
DataSec: SECTION
label1: DS.W 1
label2: DS.W 2
label3: EQU 8
codeSec: SECTION
        BASE label3
        ALIGN 4
```

## 20.1.82 A2316: Section name required

[ERROR]

### Description

A SWITCH directive is not followed by a symbol name. Absolute expressions or string are not allowed in a SWITCH directive. The symbol specified in a SWITCH directive must refer to a previously defined section.

### Example

```
dataSec: SECTION
label1: DS.B 1
codeSec: SECTION
        SWITCH $A344
```

### Tips

Specify the name of a previously define section in the SWITCH instruction.

### Example

```
dataSec: SECTION
label1: DS.B 1
codeSec: SECTION
        SWITCH dataSec
```

## 20.1.83 A2317: Illegal redefinition of section name

[ERROR]

### Description

The name associated with a section is previously used as a label in a code or data section or is specified in a XDEF directive. The macro assembler does not allow to export a section name, or to use the same name for a section and a label.

### Example

```
dataSec: SECTION
sec_Label: DS.W 3
        ; ...
sec_Label: SECTION
        ; ...
```

### Tips

Change to name of the section to a unique identifier.

### Example

```
dataSec: SECTION
dat_Label: DS.W 3
        ; ...
sec_Label: SECTION
        ; ...
```

## 20.1.84 A2318: Section not declared

[ERROR]

### Description

The label specified in a SWITCH directive is not associated with a section.

## Example

```
dataSec: SECTION
label1: DS.B 1
        ; ...
codeSec: SECTION
        ; ...
        SWITCH unknownSec
        ; ...
```

## Tips

Specify the name of a previously define section in the SWITCH instruction.

## Example

```
dataSec: SECTION
label1: DS.B 1
        ; ...
codeSec: SECTION
        ; ...
        SWITCH dataSec
        ; ...
```

### 20.1.85 A2319: No section link to this label

[ERROR]

#### Description

A label without corresponding section was detected. This error usually occurs because of other errors before.

#### Tips

Correct all errors before this one first.

## 20.1.86 A2320: Value too small

[ERROR]

### Description

The absolute expression specified in a directive is too small. This message can be generated in following cases:

- The expression specified in an ALIGN, DCB, or DS directive is smaller than 1.
- The expression specified in a PLEN directive is smaller than 10. A header is generated on the top of each page from the listing file. This header contains at least 6 lines. So a page length smaller than 10 lines does not make many sense.
- The expression specified in a LLEN, SPC or TABS directive is smaller than 0 (negative).

### Example

```

        PLEN    5

        LLEN   -4

dataSec: SECTION

        ALIGN  0

        ; ...

label1: DS.W   0

        ; ...

```

### Tips

Modify the absolute expression to a value in the range specified above.

### Example

```

        PLEN    50

        LLEN   40

dataSec: SECTION

        ALIGN  8

        ; ...

label1: DS.W   1

        ; ...

```

## 20.1.87 A2321: Value too big

[ERROR]

### Description

The absolute expression specified in a directive is too big. This message can be generated in following cases:

- The expression specified in an ALIGN directive is bigger than 32767.
- The expression specified in a PLEN directive is bigger than 10000.
- The expression specified in a LLEN directive is bigger than 132.
- The expression specified in a SPC directive is bigger than 65.
- The expression specified in a TABS directive is bigger than 128.

### Example

```
        PLEN    50000
        LLEN    200
dataSec: SECTION
        ALIGN   40000
        ; ...
```

### Tips

Modify the absolute expression to a value in the range specified above.

### Example

```
        PLEN    50
        LLEN    40
dataSec: SECTION
        ALIGN   8
        ; ...
```

## 20.1.88 A2323: Label is ignored

[DISABLE, INFORMATION, WARNING, ERROR]

## Description

A label is specified in front of a directive, which does not accept any label. The macro assembler ignores such label. These labels cannot not be referenced anywhere else in the application. Labels will be ignored in front of following directives: ELSE, ENDIF, END, ENDM, INCLUDE, CLIST, ALIST, FAIL, LIST, MEXIT, NOLIST, NOL, OFFSET, ORG, NOPAGE, PAGE, LLEN, PLEN, SPC, TABS, TITLE, TTL.

## Example

```
CodeSec: SECTION
        ; ...
label:  PLEN 50
        ; ...
label2: LIST
        ; ...
```

## Tips

Remove the label which is not required. If you need a label at that position in a section, define the label on a separate line.

## Example

```
CodeSec: SECTION
        ; ...
label:
        PLEN 50
        ; ...
label2:
        LIST
        ; ...
```

## 20.1.89 A2324: Illegal Base (2,8,10,16)

[ERROR]

## Description

An invalid base number follows a BASE directive. The valid base numbers are 2, 8, 10 or 16. The expression specified in a BASE directive must be an absolute expression and must match one of the values enumerated above.

## Example

```
        BASE  67
dataSec: SECTION
        ; ...
label:   EQU  35
        ; ...
        BASE  label
```

## Tips

Specify one of the valid value in the BASE directive.

## Example

```
        BASE  16
        ; ...
dataSec: SECTION
label:   EQU  8
        ; ...
        BASE  label
```

## 20.1.90 A2325: Comma or Line end expected

[ERROR]

## Description

An incorrect syntax has been detected in a DC, FCB, FDB, FQB, XDEF, PUBLIC, GLOBAL, XREF or EXTERNAL directive. This error message is generated when the values enumerated in one of the directive enumerated above are not terminated by an end of line character, or when they are not separated by a ',' character.

## Example

```

XDEF  dataLab1 dataLab2

XREF  bb1, bb2, bb3, bb4   This is a comment

; ...

dataSec: SECTION

dataLab1: DC.B 2 | 4 | 6 | 8

dataLab2: FCB 45, 66, 88      label3:DC.B 4

```

## Tips

Use the ',' character as separator between the different items in the list or insert an end of line at the end of the enumeration.

## Example

```

XDEF  dataLab1, dataLab2

XREF  bb1, bb2, bb3, bb4   ;This is a comment

; ...

dataSec: SECTION

dataLab1: DC.B 2, 4, 6, 8

dataLab2: FCB 45, 66, 88

label3:   DC.B 4

```

## 20.1.91 A2326: Label <Label> is redefined

[ERROR]

### Description

A label redefinition has been detected. This message is issued when:

- The label specified in front of a DS, DCB, FCC directive is already defined.
- One of the label names enumerated in a XREF directive is already defined.
- The label specified in front of an EQU directive is already defined.
- The label specified in front of a SET directive is already defined and not associated with another SET directive.
- A label with the same name as an external referenced symbol is defined in the source file.
- A label name is reused as section name.



## Example

```
Data1Sec: SECTION
label1:   DS.W   4
          ; ...

Data2Sec: SECTION
label1:   DS.W   1
          ; ...
```

## Tips

Modify your source code to use unique identifiers.

## Example

```
Data1Sec: SECTION
d1_label1: DS.W   4
          ; ...

Data2Sec: SECTION
d2_label1: DS.W   1
          ; ...
```

## 20.1.92 A2327: ON or OFF expected

[ERROR]

### Description

The syntax for a MLIST or CLIST directive is not correct. These directives expects a unique operand, which can take the value ON or OFF.

### Example

```
CodeSec: SECTION
          ; ...
          CLIST
          ; ...
```

## Tips

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014

Specify either ON or OFF after the MLIST or CLIST directive.

### Example

```
CodeSec: SECTION
; ...
CLIST ON
; ...
```

## 20.1.93 A2328: Value is truncated

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

The size of one of the constants listed in a DC directive is bigger than the size specified in the DC directive.

### Example

```
DataSec: SECTION
cst1: DC.B $56, $784, $FF
cst2: DC.W $56, $784, $FF5634
```

### Tips

Reduce the value from the constant to a value fitting in the size specified in the DC directive.

### Example

```
DataSec: SECTION
cst1: DC.B $56, $7, $84, $FF
cst2: DC.W $56, $784, $FF, $5634
```

## 20.1.94 A2329: FAIL found

[ERROR]

## Description

The FAIL directive followed by a number smaller than 500 has been detected in the source file. This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

## Example

```
LE.B:  MACRO
        IFC "\1", ""
            FAIL "no data" ; error
        MEXIT
    ENDF

        IFC "\2", ""
            FAIL 600          ; warning
        DC.B \1
        MEXIT
    ENDF

        IFNC "\3", ""
            FAIL 400          ; error
        ENDF

        DC.B \2, \1
    ENDM

LE.B    $12, $34, $56
```

### 20.1.95 A2330: String is not allowed

[ERROR]

## Description

A string has been specified as initial value in a DCB directive. The initial value for a constant block can be any byte, half-word or word absolute expression as well as a simple relocatable expression.

## Example

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014

## Assembler Messages

```
CstSec: SECTION
label: DCB.B 10, "aaaaaa"
; ...
```

### Tips

Specify the ASCII code associated with the characters in the string as initial value.

### Example

```
CstSec: SECTION
label: DCB.B 10, $61
; ...
```

## 20.1.96 A2332: FAIL found

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

The FAIL directive followed by a number bigger than 500 has been detected in the source file. This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

### Example

```
LE.B: MACRO
IFC "\1", ""
    FAIL "no data" ; error
MEXIT
ENDIF
IFC "\2", ""
    FAIL 600 ; warning
DC.B \1
MEXIT
ENDIF
IFNC "\3", ""
```

```
        FAIL 400      ; error
ENDIF
DC.B   \2,\1
ENDM
LE.B   $12
```

## 20.1.97 A2333: Forward reference not allowed

[ERROR]

### Description

A forward reference has been detected in an EQU instruction. This is not allowed.

### Example

```
CstSec: SECTION
        ; ...
equLab: EQU label2
        ; ...
label2: DC.W $6754
        ; ...
```

### Tips

Move the EQU after the definition of the label it refers to.

### Example

```
CstSec: SECTION
        ; ...
label2: DC.W $6754
        ; ...
equLab: EQU label2
        ; ...
```

## 20.1.98 A2335: Exported SET label<name> is not supported

[ERROR]

### Description

The SET directive does not allow a reference to an external label.

### Example

```
XDEF setLab
const: SECTION
lab:   DC.B   6
setLab: SET $77AA
```

### Tips

SET labels initialized with absolute expressions can be defined in a special file to be included by assembly files, or the EQU directive can be used.

### Example

```
XDEF setLab
const: SECTION
lab:   DC.B   6
setLab: EQU $77AA
```

### See also

SET Directive

## 20.1.99 A2336: Value too big

[DISABLED, INFORMATION, WARNING, ERROR]

### Description

The absolute expression specified as initialization value for a block defined using DCB is too big. This message is generated when the initial value specified in a DCB.B directive cannot be coded on a byte. In this case the value used to initialize the constant block will be truncated to a byte value.

### Example

```
constSec: SECTION
        ; ...
label1:  DCB.B  2, 312
        ; ...
```

In the previous example, the constant block is initialized with the value \$38 (= 312 & & \$FF)

### Tips

To avoid this warning, modify the initialization value to a byte value.

### Example

```
constSec: SECTION
        ; ...
label1:  DCB.B  2, 56
        ; ...
```

## 20.1.100 A2338: <FailReason>

[ERROR]

### Description

The FAIL directive followed by a string has been detected in the source file. This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly, to detect user defined error or warning condition

### Example

```
LE.B:   MACRO
        IFC "\1", ""
        FAIL "no data" ; error
```

```

MEXIT
ENDIF
IFC "\2", ""
    FAIL 600          ; warning
    DC.B \1
MEXIT
ENDIF
IFNC "\3", ""
    FAIL 400          ; error
ENDIF
DC.B \2, \1
ENDM
LE.B      ; no args

```

### 20.1.101 A2340: Macro parameter already defined

[ERROR]

#### Description

A name of a macro parameter was already defined.

#### Note

Not all assemblers do support named macro parameters. Assembler not supporting this will never issue this message.

### 20.1.102 A2341: Relocatable Section Not Allowed: an Absolute file is currently directly generated

[ERROR]

#### Description

A relocatable section has been detected while the assembler tries to generate an absolute file. This is not allowed.



## Example

```
DataSec: SECTION
Data1: DS.W 1
        ORG $800
entry:
        NOP
        NOP
```

### 20.1.103 A2342: Label in an OFFSET section cannot be exported

[ERROR]

#### Description

An external defined label is provided as offset in an OFFSET directive or a label defined in an offset is used in a DS directive.

#### Example

```
        OFFSET 1
ID:     DS.B 1
        ALIGN 4
COUNT: DS.W 1
        ALIGN 4
VALUE:  DS.W 1
SIZE:   EQU *
        XDEF VALUE
DataSec: SECTION
Struct: DS.B SIZE
```

#### Tips

Use other labels to specify the size of the offset and the number of space to provide.

#### Example

```
        OFFSET 1
```

## Assembler Messages

```

ID:      DS.B  1
        ALIGN 4

COUNT:  DS.W  1
        ALIGN 4

VALUE:   DS.W  1

SIZE:    EQU  *

DataSec: SECTION

Struct:  DS.B  SIZE

```

### 20.1.104 A2345: Embedded type definition not allowed

[ERROR]

#### Description

The keyword STRUCT or UNION has been detected within a structured type definition. This is not allowed.

#### Note

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

#### Example

```

myType: STRUCT
field1:  DS.W  1
field2:  DS.W  1
field3:  DS.B  1
fieldx:  STRUCT
    xx:   DS.B  1
    yy:   DS.B  1
        ENDSTRUCT
field4:  DS.B  3
field5:  DS.W  1
        ENDSTRUCT

```

#### Tips

Define the structured type as two separate structured types.

### Example

```
typeX:  STRUCT
    xx:   DS.B 1
    yy:   DS.B 1
        ENDSTRUCT

myType: STRUCT
    field1: DS.W 1
    field2: DS.W 1
    field3: DS.B 1
    fieldx: TYPE typeX
    field4: DS.B 3

    field5: DS.W 1
        ENDSTRUCT
```

## 20.1.105 A2346: Directive or instruction not allowed in a type definition

[ERROR]

### Description

An instruction or an invalid directive has been detected in a structured type definition. Only following directives are allowed in a structured type definition:

- DS, RMB, ALIGN, EVEN, LONGEVEN
- Conditional Assembly directives (IF, ELSE, IFCC, ..)
- Directives related to the formatting of the listing file (PLEN, SPC, ...)
- XDEF, XREF, BASE

Not all assembler backends do support structured types. Assembler not supporting them will not issue this message.

### Example

```
myType: STRUCT
    field1: DS.W 1
```

## Assembler Messages

```
field2: DS.W 1
field3: DS.B 1
cst: DC.B $34
field4: DS.B 3
field5: DS.W 1
        ENDSTRUCT
```

### Tips

Remove the invalid directive or instruction.

### Example

```
myType: STRUCT
field1: DS.W 1
field2: DS.W 1
field3: DS.B 1
field4: DS.B 3
field5: DS.W 1
        ENDSTRUCT
```

## 20.1.106 A2350: MEXIT is illegal (detected outside of a macro)

[ERROR]

### Description

An MEXIT was found without a matching MACRO directive.

### Example

```
MEXIT
```

### Tips

Check for the correct writing of the MACRO directive. Do not use MEXIT as label.

## 20.1.107 A2351: Expected Comma to separate macro arguments

[DISABLE, INFORMATION, **WARNING** , ERROR]

### Description

Macro arguments must be separated by a comma.

### Example

```
constants MACRO
    DC.B \1+1, \2+1
ENDM
constants 1 2
```

### Tips

Do not use spaces in macro parameters, instead use a comma:

```
constants 1,2
```

## 20.1.108 A2352: Invalid Character

[ERROR]

### Description

An invalid character was found during parsing.

### Tips

Check the source file for binary parts.

## 20.1.109 A2353: Illegal or unsupported directive SECT

[DISABLED, INFORMATION, WARNING, **ERROR** ]

### Description

The assembler did not understand the whole SECT directive. The SECT directive is only recognized when the option Option -Compat is present.

### Tips

Use the SECTION directive instead.

### See also

Option -Compat

## 20.1.110 A2354: Ignoring directive '<directive>'

[DISABLED, INFORMATION, **WARNING** , ERROR]

### Description

The assembler is ignoring the specified directive. This message is used mainly for directives which are not supported when the option Option -Compat is present.

### See also

Option -Compat

## 20.1.111 A2355: Illegal size specification

[DISABLED, INFORMATION, WARNING, **ERROR** ]

### Description

The size specification given is not legal for this directive.

### Tips

Use no size specification at all or use a different one.

## 20.1.112 A2356: Illegal RAD50 character

[DISABLED, INFORMATION, WARNING, **ERROR** ]

### Description

**Note**

Not all assemblers do support the RAD50 directive. This message is only issued by assemblers which do support the RAD50 directive.

**See also**

Directive RAD50

**20.1.113 A2357: Illegal macro argument '<Argument>'**

[DISABLED, INFORMATION, WARNING, ERROR]

**Description**

Macro argument started with the [? syntax have to end with ?]. However this second pattern was not found.

**See also**

- Macro argument grouping
- Macros chapter
- Option -CMacAngBrack

**20.1.114 A2358: Size prefix ignored**

[DISABLED, INFORMATION, WARNING, ERROR]

**Description**

The used size prefix did not match with an previously specifid size prefix and was therefore ignored.

**Tips**

Don't specify a size prefix unless you are certain about it. Use always the same prefix for the same label.

**20.1.115 A2359: String index out of bounds**

[DISABLED, INFORMATION, WARNING, **ERROR**]

### Description

The syntax is SUBSTR('STRING',start,count). The message occurs if 'start' or 'count' value is less than zero.

## 20.1.116 A2360: Maximum SUBSTR recursion level reached

[DISABLED, INFORMATION, WARNING, **ERROR**]

### Description

This message occurs when the recursive depth of SUBSTR operator (for example: SUBSTR(SUBSTR(..))) exceeds the count 3000.

## 20.1.117 A2380: Cutting very long line

[DISABLED, INFORMATION, WARNING, **ERROR**]

### Description

A line was longer than the limit 1024 characters. All remaining text is ignored.

### Tips

- Split up the line into several lines.
- Remove trailing spaces and tabs.
- Use shorter identifiers.

## 20.1.118 A2381: Previous message was in this context '<Context>'

[DISABLED, INFORMATION, WARNING, **ERROR**]

### Description

The previous message was in a special context. Usually this message is used to show the current macro expansion tree.



## Example

```
TABLE: MACRO
    ; \1: size of table to be generated
    ; \2: current value for table
    \@size: EQU \1
    if (\@size >= 2)
        TABLE \@size/2,\2
        TABLE \@size-\@size/2,\2+\@size/2
    else
        if (\@size == 1)
            DC \2
        endif
    endif
ENDM
TABLE 4
```

Generates the following messages:

```
b.asm(9): ERROR A1055: Error in expression
INFORMATION Macro Expansion          DC
b.asm(5): INFORMATION A2381: Previous message was in this
context 'Macro Invocation'
```

So the error happens at line 9 ("DC \2") which was called by line 5 twice and finally by line 14. To fix this example, add a second parameter to the TABLE macro call:

```
TABLE 4,0
```

## Tips

Check the message before the first A2381 to see the cause of the problem.

### 20.1.119 A2382: Illegal character ('\0') in source file

[ERROR]

## Description

An zero byte (a byte with ASCII code 0) was found in the source.

## Tips

Check if the source file is binary.

## 20.1.120 A2383: Input line too long

[ERROR]

## Description

An input line is longer then the translation limit. Input lines must not be longer than 1024 characters.

## Tips

Split the input line. In recursive macros, use local SET labels to avoid lines growing with the input buffer: Instead of:

```
TableTo: MACRO
    if (\1 > 0)
        DC.W \1
        TableTo \1 - 1
    endif
ENDM
```

Use:

```
TableTo: MACRO
    if (\1 > 0)
        DC.W \1
        \@LocLabel: SET \1-1
        TableTo \@LocLabel
    endif
ENDM
```

## 20.1.121 A2400: End of Line expected

[DISABLED, INFORMATION, WARNING, **ERROR** ]

### Description

The assembler did not expect anything anymore on a line. This message can be generated when: A comment, which does not start with the start of comment character (';'), is specified after the instruction. A further operand is specified in the instruction. ...

### Example

```
DataSec: SECTION
var:    DS.B 1    Char variable
```

### Tips

Remove the invalid character or sequence of characters from the line. Insert the start of comment character at the beginning of the comment. Remove the superfluous operand.

### Example

```
DataSec: SECTION
var:    DS.B 1    ; Char variable
```

## 20.1.122 A2401: Complex relocatable expression not supported

[ERROR]

### Description

A complex relocatable expression has been detected. A complex relocatable expression is detected when the expression contains:

- An operation between labels located in two different sections.
- A multiplication, division or modulo operation between two labels.
- The addition of two labels located in the same section.

### Example

```

        XDEF offset
DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataSec2: SECTION SHORT
DataLbl2: DS.W 15
offset:  EQU DataLbl2 - DataLbl1

```

## Tips

The macro assembler does not support complex relocatable expressions. The corresponding expression must be evaluated at execution time.

## Example

```

DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataSec2: SECTION SHORT
DataLbl2: DS.W 15
Offset:   DS.W 1
...
CodeSec: SECTION
...
evalOffset:
    LD   A, #DataLbl2
    SUB  A, #DataLbl1
    LD   Offset, A
        XDEF  Offset

```

## Example

```

DataSec1: SECTION SHORT
DataLbl1: DS.B 10
DataSec2: SECTION SHORT
DataLbl2: DS.W 15
Offset:   DS.W 1
; ...
CodeSec: SECTION
; ...

```

```

evalOffset:
    LD    A,#%lo(DataLb12)
    SUB  A,#%lo(DataLb11)
    LD    Offset+1,A
    LD    A,#%hi(DataLb12)
    SBC  A,#%hi(DataLb11)
    LD    Offset+0,A
    
```

## Example

```

DataSec1: SECTION SHORT
DataLb11: DS.B 10
DataSec2: SECTION SHORT
DataLb12: DS.W 15
Offset:   DS.W 1
...
CodeSec: SECTION
...
evalOffset:
    LDA  #DataLb12
    SUB  #DataLb11
    STA  Offset
    
```

## Example

```

DataSec1: SECTION SHORT
DataLb11: DS.B 10
DataSec2: SECTION SHORT
DataLb12: DS.W 15
Offset:   DS.W 1
...
CodeSec: SECTION
...
evalOffset:
    LDD  #DataLb12
    SUBD #DataLb11
    
```

```

STD  Offset
      XDEF Offset

```

## Example

```

DataSec1: SECTION SHORT
DataLb11: DS.B 10
DataSec2: SECTION SHORT
DataLb12: DS.W 15
Offset:   DS.W 1
          ; ...
CodeSec:  SECTION
          ; ...
evalOffset:
      LI   R4,DataLb11
      LI   R5,DataLb12
      SUBF R4,R4,R5
      STH  R4,Offset

```

## Example

```

DataSec1: SECTION
DataLb11: DS.B 10
DataSec2: SECTION
DataLb12: DS.W 15
Offset:   DS.W 1
...
CodeSec:  SECTION
...
evalOffset:
      LRW R6,[addLb11]
      LD.B R7,(R6,0)
      LRW R6,[addLb11]
      LD.B R7,(R6,0)
      SUBU R7,R8
      LRW R6,[addOff]

```

```
LD.B R7, (R6, 0)

ALIGN 4

addLb11: DC.W DataLb11
addLb11: DC.W DataLb12
addOff:  DC.W Offset
```

## Example

```
DataSec1: SECTION SHORT
DataLb11: DS.B 10
DataSec2: SECTION SHORT
DataLb12: DS.W 15
Offset:   DS.W 1
CodeSec:  SECTION

MOVE    #DataLb12, A0
SUB     #DataLb11, A0
MOVE    A0, Offset
```

If both DataSec1 and DataSec2 are in the same section and defined in this module, the assembler can compute the difference:

```
DataSec1: SECTION SHORT
DataLb11: DS.B 10
DataLb12: DS.W 15
offset: EQU DataLb12 - DataLb11
```

### 20.1.123 A2402: Comma expected

[ERROR]

#### Description

A comma character is missing between two operands of an instruction or directive.

#### Example

## Assembler Messages

```
DataSec: SECTION
Data:    DS.B 1
ConstSec: SECTION
         DC.B 2 3
```

### Tips

The comma (',') character is used as separator between instruction operands.

### Example

```
DataSec: SECTION
Data:    DS.B 1
ConstSec: SECTION
         DC.B 2, 3
```

## 20.1.124 A2500: Equal expected

[ERROR]

### Description

In a for directive, a = was expected.

### Example

```
FOR j := $1000 TO $1003
    DC.W j
ENDFOR
```

### Tips

Just use an equal in the example (no colon).

```
FOR j = $1000 TO $1003
    DC.W j
ENDFOR
```

Check that the Option -Compat=b is enabled.



**See also**

- Option -Compat
- Directive FOR

**20.1.125 A2501: TO expected**

[ERROR]

**Description**

In a for directive, a TO was expected.

**Example**

```
FOR j := $1000 < $1003
DC.W j
ENDFOR
```

**Tips**

Just use a TO in the example.

```
FOR j = $1000 TO $1003
DC.W j
ENDFOR
```

Check that the Option -Compat=b is enabled.

**See also**

- Option -Compat
- Directive FOR

**20.1.126 A2502: ENDFOR missing**

[ERROR]

**Description**

In a for directive, a ENFOR was expected.

## Example

```
FOR j := $1000 < $1003
DC.W j
```

## Tips

Check that every FOR has a corresponding ENDFOR.

```
FOR j = $1000 TO $1003
DC.W j
ENDFOR
```

Check that the Option -Compat=b is enabled.

## See also

- Option -Compat
- Directive FOR

## 20.1.127 A2503: ENDFOR without FOR

[ERROR]

### Description

A ENDFOR without corresponding FOR was found.

### Example

```
; FOR j := $1000 < $1003
DC.W j
ENDFOR
```

## Tips

Check that every ENDFOR has a corresponding FOR. In the example, remove the semicolon.

```
FOR j = $1000 TO $1003
DC.W j
```

ENDFOR

Check that the Option -Compat=b is enabled.

**See also**

- Option -Compat
- Directive FOR

### **20.1.128 A2600: Addition of values from a different address space**

[ERROR]

**Description**

Addition of two values, that are not in the same address space.

### **20.1.129 A2601: Subtraction of values from a different address space**

[ERROR]

**Description**

Subtraction of two values, that are not in the same address space.

### **20.1.130 A2602: Operation of values from a different address space**

[ERROR]

**Description**

Operation of two values, that are not in the same address space.

## 20.1.131 A3000: User requested stop

[DISABLE, INFORMATION, WARNING, **ERROR** ]

### Description

This message is used when the user presses the stop button in the graphical user interface. Also when the assembler is closed during an assembly, this message is issued.

### Tips

By moving this message to a warning or less, the stop functionality can be disabled.

## 20.1.132 A4000: Recursive definition of label <Label name>="">

[ERROR]

### Description

The definition of an EQU label depends directly or indirectly on itself.

### Example

```
XDEF tigger

pooh: EQU tigger - 2

tigger: EQU 2*pooh
```

### Tips

This error usually indicates an error in some definitions. Determine the labels involved in the recursive definition and eliminate the circular dependency.

## 20.1.133 A4001: Data directive contains no data

[WARNING]

## Description

A data directive is empty, and no code is generated for this directive.

## Example

```
DC.B      ; 1,2,3,4
```

## Tips

This warning may indicate an error, or it may be intentional within a macro expansion, for example.

### 20.1.134 A4002: Variable access size differs from previous declaration

[WARNING]

#### Description

An implicit or explicit declaration of a label indicates an access size which differs from a former declaration.

#### Tips

Indicating the access size of variables is particularly helpful in "header" files which contain XREF directives, to be included by other files accessing these variables. If an assembly file contains a

```
XREF.B obj
```

this warning message indicates potential problems.

### 20.1.135 A4003: Found XREF, but no XDEF for label <Label>, ignoring XREF

[WARNING]

#### Description

The local definition of a label <Label> supersedes a global XREF declaration, if no appropriate XDEF directive is given.

### Example

```

XREF main

Code: SECTION

main: NOP ; is local, unless XDEF given

NOP

```

### Tips

This warning may indicate a forgotten XDEF directive.

## 20.1.136 A4004: Qualifier ignored

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

An unknown qualifier to a SECTION or ORG directive is ignored.

### Example

```

const: SECTION SHORT 1234 FOO
      DC.B "hello", 0

```

### Tips

This warning may indicate a misspelled qualifier.

## 20.1.137 A4005: Access size mismatch for <Symbol>

[WARNING]

## Description

Incompatible access sizes are attached to an object, either implicitly or explicitly. The access size of an object is determined from XREF declarations, XDEF definitions and (if applicable) from the access size of the section, where the object is placed into.

## Example

```
XDEF.B two

const: SECTION

two: DC.B 2 ; implicit *.W definition
```

## Tips

It is probably a good idea to eliminate mismatches, particularly if mismatches occur between declarations in a "header file" and definitions in the assembly file.

### 20.1.138 A4006: Illegal value `<ErrorDescription>'

[DISABLE, INFORMATION, WARNING, **ERROR** ]

## Description

The assembler could not compute the value of a constant. One possible reason for this failure is an address space conversion function which is called with an address which is illegal for the source address space.

## Tips

Check the documentation for the used functions and fixup types.

### 20.1.139 A4100: Address space clash for <Symbol>

[ERROR]

## Description

This message is only relevant for Harvard architectures (separate code and data address spaces), and occurs for symbols whose address is used both as a code address and a data address.

## Tips

This clash may be intentional, but indicates an error in most cases.

### 20.1.140 A13001: Illegal Addressing Mode. <AddrModes>

[ERROR]

## Description

An illegal addressing mode has been detected in an instruction. This message is generated when an incorrect encoding is used for an addressing mode.

## Example

```
STA # $45
```

## Tips

Use a valid notation for the addressing mode encoding.

## Example

```
STA $45
```

### 20.1.141 A13003: Value is truncated to one byte

[DISABLED, INFORMATION, WARNING, ERROR]

## Description

An operand is only possible as 8 bit direct mode, but the actual passed in one is larger. As labels are 16 bit by default.



## Example

```
MyData: SECTION
data:   DS.B 1
POS     EQU 2
MyCode: SECTION
entry:
    ADD data
    BCLR 7, data
    BRCLR 4, data, entry
    BSET 3, data
    BRSET POS, data, entry
```

The following instructions do only support direct memory access. Using them with a label without explicit size specification causes this message.

```
XREF label
ASR label
BCLR 2,label
BRSET 0, label, *
BSET 0,label
CBEQ label,*
CLR label
COM label
CPHX label; legal for HCS08
DBNZ label, *
DEC label
INC label
LDHX label; legal for HCS08
LSL label
LSR label
MOV #1, label
MOV label, label
NEG label
ROL label
ROR label
```

## Assembler Messages

```
STHX label; legal for HCS08
TST label
```

## Tips

Use the Direct addressing Mode for the operand: as an example, you can define data in a section located on page zero using the qualifier `SHORT` after the `SECTION` directive. For labels defined with `XREF`, use the `XREF.B` or `XREFB` directive. For labels defined with `EQU`, use `EQU.B` if the right side cannot be automatically determined to be direct.

```
MyData: SECTION SHORT
data: DS.B 1
POS EQU 2
MyCode: SECTION
entry:
    CPHX data
    BCLR 7, data
    BRCLR 4, data, entry
    BSET 3, data
    BRSET POS, data, entry
```

## Example

```
XREF.B label
ASR label
BCLR 2,label
```

## See also

- chapter Using Direct Addressing mode to access Symbols

## 20.1.142 A13004: Value is truncated to two bytes

[DISABLED, INFORMATION, WARNING, ERROR]

### Description

The value is larger than 16 bits. The higher bits are ignored.

## Tips

Specify only the lower 16 bits. Use the ">" operator to cut larger expressions.

### 20.1.143 A13101: Illegal operand format

[ERROR]

#### Description

An operand used in the instruction is using an invalid addressing mode.

#### Example

As an example, the following code generates the A13101 error message.

```
Entry:
    ADC X+
    LDA 2, SP
```

## Tips

To solve this problem, use an allowed addressing mode for the instruction.

```
Entry:
    ADC ,X
    ADC X
    ADC #$5
    LDA 2, X
```

### 20.1.144 A13102: Operand not allowed

[ERROR]

#### Description

This error message is issued for instruction BCLR or BRSET when the operand is not a DIRECT or an EXTENDED.

#### Example

CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08/RS08 Assembler Reference Manual,  
Rev. 10.6, 02/2014

Entry:

```
BRCLR 7, X
BRCLR 7, SP
BSET 7, X
```

## Tips

To solve this problem, use an allowed addressing mode for the instruction.

### 20.1.145 A13106: Illegal size specification

[ERROR]

#### Description

A size operator follows an instruction. Size operators are coded as semicolon character followed by single character.

#### Example

```
MyData: SECTION
data: DS.B 1
MyCode: SECTION
entry:
    ADC.B data
    ADC.L data
    ADC.W data
    ADC.b data
    ADC.l data
    ADC.w data
```

## Tips

Remove the size specification following the instruction.

#### Example

```
MyData: SECTION
data: DS.B 1
```

```
MyCode: SECTION
entry:
    ADC data
```

## 20.1.146 A13109: Positive value expected

[ERROR]

### Description

When using the instruction BSET, BCLR, BRSET and BRCLR, this error message is issued if the specified value for the bit number is negative.

### Example

```
MyData: SECTION SHORT
data:   DS.B 1
NEG     EQU -2
MyCode: SECTION
entry:
    BCLR  -7, data
    BRCLR -4, data, entry
    BSET  -3, data
    BRSET NEG, data, entry
```

### Tips

Use a positive value for the bit number:

```
MyData: SECTION SHORT
data:   DS.B 1
POS     EQU 2
MyCode: SECTION
entry:
    BCLR  7, data
    BRCLR 4, data, entry
    BSET  3, data
```

```
BRSET POS, data, entry
```

## 20.1.147 A13110: Bit number expected

[ERROR]

### Description

When using the instruction BSET, BCLR, BRSET and BRCLR, this error message is issued if the specified value for the bit number is not a Direct or an Extended.

### Example

```
MyData: SECTION SHORT
data: DS.B 1
MyCode: SECTION
entry:
    BCLR #$7, data
    BRCLR #$4, data, entry
    BRSET #$3, data, entry
    BSET #$2, data
```

### Tips

Use a correct value for the bit number: 0, 1, 2, 3, 4, 5, 6, 7

```
MyData: SECTION SHORT
data: DS.B 1
MyCode: SECTION
entry:
    BCLR 7, data
    BRCLR 4, data, entry
    BSET 3, data
    BRSET 2, data, entry
```

## 20.1.148 A13111: Value out of range

[DISABLE, INFORMATION, WARNING, ERROR]

### Description

When using the instruction BSET, BCLR, BRSET and BRCLR, this error message is issued if the specified value for the bit number is greater than 7.

### Example

```
MyData: SECTION SHORT
data:   DS.B 1
MyCode: SECTION
entry:
    BCLR    20, data
    BRCLR   70, data, entry
    BSET    9, data
    BRSET  200, data, entry
```

### Tips

Use a correct value for the bit number: 0, 1, 2, 3, 4, 5, 6, 7

### Example

```
MyData: SECTION SHORT
data:   DS.B 1
MyCode: SECTION
entry:
    BCLR    7, data
    BRCLR   4, data, entry
    BSET    3, data
    BRSET   2, data, entry
```

## 20.1.149 A13203: Not a HC08 instruction or directive

[ERROR]

### Description

The identifier detected in an assembly line instruction part is neither an assembly directive, nor an HC08 instruction, nor a user defined macro.

### Example

```
CodeSec: SECTION
...
LDAA #$5510
```

### Tips

- Change the identifier to an assembly directive, an HC08 instruction, or the name of a user defined macros.
- If you are using a directive: make sure that there is at least one space in front of the directive.

## 20.1.150 A13204: Instruction not supported by RS08 Core

[DISABLE, INFORMATION, WARNING, **ERROR** ]

### Description

The instruction specified does exist for HC08 (or HCS08) but not for the RS08 which only supports a subset of the full HC08 instruction set.

### Example

```
CodeSec: SECTION
...
PSHX
```

### Tips

- Check the RS08 reference manual for the supported instructions.

## 20.1.151 A13205: RS08 instructions only supported in RS08 mode (use option -Crs08)



[ERROR]

### Description

The instruction exists for the RS08 core, but not for the HC08 (or HCS08) core. As the assembler is currently in HC08 mode, it does not support it.

### Example

```
CodeSec: SECTION
    ADCX
```

### Tips

- Use the option `-Crs08` to compile RS08 code.
- When adapting RS08 code for the HC08, check the reference manual which instructions exist.

## 20.1.152 A13206: This instruction is only available for derivatives with MMU

[ERROR]

### Description

This message occurs for 'CALL' and 'RTC' instructions.

### Tips:

- Specify `-MMU` option along with `-Cs08` option (HCS08 derivative).



# Index

- '<cmd>' [419](#)
- '<Context>' [496](#)
- '<Description>' [459](#)
- '<ErrorDescription>' [511](#)
- '<file>' [419](#)
- '<NewValue>'. [420](#)
- '<OldValue>', [420](#)
- '<Symbol>' [454](#)
- '<Argument>' [495](#)
- '<description>' [422](#)
- '<directive>' [494](#)
- '<file>' [418](#)
- "err.log" [202](#)
- (\0) [497](#)
- (<LabelName>): [460](#)
- (2,8,10,16) [478](#)
- (Assembler) [146](#)
- (available) [191](#)
- (binary) [268–272](#)
- (detected) [492](#)
- (-Fh, [166](#)
- (HC(S)08) [248](#)
- (Integers) [411](#)
- (MMU) [192](#)
- (-Ms, [189](#)
- (PC) [126](#)
- (RS08) [259, 260](#)
- (Set) [426](#)
- (shared) [105, 106](#)
- (Test1.asm) [384](#)
- (Test1.inc) [384](#)
- (Test2.asm) [385](#)
- (Test2.prm) [385](#)
- (unary) [269, 271, 272, 276](#)
- (use) [520](#)
- (usually) [126](#)
- (-WmsgFbv, [208](#)
- (-WmsgFiv, [210](#)
- [<LowValue>..<HighValue>] [457](#)
- [Editor] [397, 401](#)
- [Installation] [393](#)
- [Options] [394](#)
- [XXX\_Assembler] [395, 403](#)
- <:> [459](#)
- <Address> [456](#)
- <AddrModes> [512](#)
- <Character> [461](#)
- <Description> [458](#)
- <details> [438](#)
- <FailReason> [487](#)
- <FieldName> [442](#)
- <FileName> [420](#)
- <kind> [418](#)
- <Label> [508](#)
- <Label> [437, 480](#)
- <Label>, [509](#)
- <LabelName> [462](#)
- <MessageId> [419](#)
- <name> [438](#)
- <Name> [422](#)
- <Option> [420](#)
- <OptionName>. [420](#)
- <Size> [456](#)
- <SizeLimit> [439](#)
- <Symbol> [435, 460, 510, 511](#)
- <Symbol>. [436](#)
- <TypeName> [439, 441](#)
- <variablename> [422](#)
- 128..127 [448](#)
- 32768..32767 [450](#)
- ArgFile: [154](#)
- Asmdbg [95](#)
- AsmDbg: [155](#)
- Ci [91](#)
- Ci: [155](#)
- CMacAngBrack [92](#)
- CMacAngBrack: [156](#)
- CMacBrackets [92](#)
- CMacBrackets: [157](#)
- Compat: [158](#)
- Compat= [94](#)
- Compat=! [94](#)
- Compat== [94](#)
- Compat=a [93](#)
- Compat=b [93](#)
- Compat=c [93](#)
- Compat=f [93](#)
- Compat=s [93](#)
- Crs08) [520](#)
- CS08/-C08/-CRS08: [161](#)
- D [91](#)
- D: [162, 459](#)
- DefLabel: [164](#)
- Env [95](#)
- Env: [165](#)
- F [89](#)
- F2, [166](#)
- F2o, [166](#)
- FA1 [429](#)
- FA2 [429](#)
- FA2): [166](#)
- FA2o, [166](#)
- H: [168](#)
- I: [169](#)
- I [91](#)
- L [89](#)
- L: [169](#)

- Lasmc
  - Lasmc=a [79, 90](#)
  - Lasmc=c [79, 90](#)
  - Lasmc=i [79, 90](#)
  - Lasmc=k [79, 90](#)
  - Lasmc=l [79, 90](#)
  - Lasmc=m [79, 90](#)
  - Lasmc=r [79, 90](#)
  - Lasmc=s [79](#)
- Lasmc: [172](#)
- Lasmc=s [90](#)
- Lasms: [174](#)
- Lc [89](#)
- Lc: [175](#)
- Ld [89](#)
- Ld: [177](#)
- Le [89](#)
- Le: [180](#)
- Li [89](#)
- Li: [182](#)
- Lic: [184](#)
- LicA: [185](#)
- LicBorrow [95](#)
- LicBorrow: [186](#)
- LicWait [95](#)
- LicWait: [187](#)
- Ll: [188](#)
- MacroNest [92](#)
- MacroNest: [190](#)
- MacroNest) [426](#)
- MCUasm [98](#)
- MCUasm: [191](#)
- MMU [98](#)
- MMU: [192](#)
- Mt): [189](#)
- N: [192](#)
- NoBeep: [193](#)
- NoDebugInfo: [194](#)
- NoEnv: [195](#)
- ObjN: [196](#)
- Prod: [197](#)
- Struct [91](#)
- Struct: [198](#)
- V: [199](#)
- View: [200](#)
- W1 [96](#)
- W1: [201](#)
- W2 [96](#)
- W2: [201](#)
- WErrFile: [202](#)
- Wmsg8x3 [96](#)
- Wmsg8x3: [203](#)
- WmsgCE: [204](#)
- WmsgCF: [205](#)
- WmsgCI: [206](#)
- WmsgCU: [207](#)
- WmsgCW: [207](#)
- WmsgFb [208](#)
- WmsgFbm): [208](#)
- WmsgFi [210](#)
- WmsgFim): [210](#)
- WmsgFob [97](#)
- WmsgFob: [212](#)
- WmsgFoi: [213](#)
- WmsgFonf [97](#)
- WmsgFonf: [215](#)
- WmsgFonp [97](#)
- WmsgFonp: [217](#)
- WmsgNe [97](#)
- WmsgNe: [218](#)
- WmsgNi [97](#)
- WmsgNi: [219](#)
- WmsgNu: [220](#)
- WmsgNw [97](#)
- WmsgNw: [222](#)
- WmsgSd: [223](#)
- WmsgSe: [223](#)
- WmsgSi: [224](#)
- WmsgSw: [225](#)
- WOutFile: [226](#)
- WStdout: [227](#)
  
- 16-bit [253, 255](#)
- 8.3 [203](#)
- 8-bit [252, 254, 258](#)
  
- A**
- A1: [417](#)
- A1000: [422](#)
- A1001: [424](#)
- A1002: [424](#)
- A1003: [425](#)
- A1004 [191](#)
- A1004: [426](#)
- A1051: [427](#)
- A1052: [428](#)
- A1053: [429](#)
- A1054: [429](#)
- A1055: [430](#)
- A1056: [431](#)
- A1057: [432](#)
- A1058: [432](#)
- A1059: [432](#)
- A1060: [433](#)
- A1061: [433](#)
- A1062: [433](#)
- A1101: [434](#)
- A1103: [434](#)
- A1104: [435](#)
- A1105: [436](#)
- A1106: [436](#)
- A1201: [437](#)
- A1202: [438](#)

A1203:	438	A2313:	472
A1251:	438	A2314:	472
A1252:	438	A2316:	473
A1253:	439	A2317:	474
A13001:	512	A2318:	474
A13003:	512	A2319:	475
A13004:	514	A2320:	476
A1301:	439	A2321:	477
A1302:	441	A2323:	477
A1303:	441	A2324:	478
A1304:	442	A2325:	479
A1305:	445	A2326:	480
A13101:	515	A2327:	481
A13102:	515	A2328:	482
A13106:	516	A2329:	482
A13109:	517	A2330:	483
A13110:	518	A2332:	484
A13111:	519	A2333:	485
A13203:	519	A2335:	486
A13204:	520	A2336:	486
A13205:	520	A2338:	487
A13206:	521	A2340:	488
A1401:	448	A2341:	488
A1402:	450	A2342:	489
A1405:	450	A2345:	490
A1406:	451	A2346:	491
A1407:	452	A2350:	492
A1408:	452	A2351:	493
A1410:	452	A2352:	493
A1411:	453	A2353:	493
A1412:	454	A2354:	494
A1413:	455	A2355:	494
A1414:	455	A2356:	494
A1415:	456	A2357:	495
A1416:	456	A2358:	495
A1417:	457	A2359:	495
A1418:	458	A2360:	496
A1419:	458	A2380:	496
A1502:	459	A2381:	496
A1503:	459	A2382:	497
A1601:	459	A2383:	498
A1602:	460	A2400:	499
A1603:	460	A2401:	499
A1604:	461	A2402:	503
A1605:	462	A2500:	504
A2:	418	A2501:	505
A2301:	462	A2502:	505
A2302:	463	A2503:	506
A2303:	464	A2600:	507
A2304:	465	A2601:	507
A2305:	466	A2602:	507
A2306:	466	A3000:	508
A2307:	467	A4000:	508
A2308:	468	A4001:	508
A2309:	469	A4002:	509
A2310:	469	A4003:	509
A2311:	470	A4004:	510
A2312:	471	A4005:	510

A4006: [511](#)  
 A4100: [511](#)  
 A50: [418](#)  
 A51: [419](#)  
 A52: [419](#)  
 A53: [419](#)  
 A54: [420](#)  
 A56: [420](#)  
 A64: [420](#)  
 A65: [422](#)  
 A66: [422](#)  
 ABS [69](#)  
 Abs. [344](#)  
 ABSENTRY [287, 437](#)  
 absolute [236, 321, 365, 369, 371, 381, 437, 454, 456, 472](#)  
 Absolute [69, 70, 74, 130, 144, 231, 278, 456, 488](#)  
 ABSPATH: [130](#)  
 access [386, 387, 509](#)  
 Access [510](#)  
 Accessing [355, 362, 363](#)  
 Adapting [70](#)  
 Adding [37](#)  
 Addition [268, 507](#)  
 address [174, 362, 507](#)  
 Address [511](#)  
 addressing [386, 387, 452, 453](#)  
 Addressing [248, 259, 512](#)  
 Align [288](#)  
 ALIGN [288](#)  
 alignment [301, 313](#)  
 all [105, 106, 287](#)  
 allocation [164](#)  
 allowed [360, 424, 429, 452, 454, 459, 483, 485, 490, 491, 515](#)  
 Allowed: [488](#)  
 already [488](#)  
 Already [119](#)  
 Analysis [35](#)  
 Analyzing [42](#)  
 Angle [156](#)  
 another [310](#)  
 ANSI-C [355, 356](#)  
 Appendices [391](#)  
 application [371, 373, 383](#)  
 Application [60, 200, 287](#)  
 applications [365, 366](#)  
 Applications [353, 365](#)  
 are [429, 452, 459](#)  
 area. [234](#)  
 areas [235](#)  
 argument [338, 436, 495](#)  
 arguments [157, 493](#)  
 Arguments [156](#)  
 ASMOPTIONS: [130](#)  
 Assemble [119, 120](#)  
 Assembled [119](#)  
 assembler [130, 287](#)

Assembler [25, 27, 47, 48, 74–77, 79–81, 83, 84, 86–90, 92–95, 97–100, 103, 104, 121, 149, 152, 199, 239, 283, 343, 353, 417](#)  
 Assembling [46, 47, 54, 119](#)  
 assembly [155, 290, 296, 297, 299, 305, 307, 308, 355, 356, 366, 369, 385](#)  
 Assembly [30, 41, 54, 69, 70, 74, 286, 365, 384](#)  
 Assembly-Control [285](#)  
 Associated [117](#)  
 attributes [229](#)  
 available [187, 521](#)

## B

Bar [102, 103](#)  
 base [289](#)  
 Base [478](#)  
 BASE [289](#)  
 batch [208, 212](#)  
 Batch [146](#)  
 because [432](#)  
 been [119](#)  
 beep [193](#)  
 Begin [314](#)  
 beginning [461](#)  
 big [477, 486](#)  
 Binary [280](#)  
 Bit [518](#)  
 Bitwise [271](#)  
 Blank [330](#)  
 block [298, 305, 424](#)  
 Block [294](#)  
 Borrow [186](#)  
 bounds [495](#)  
 box [192](#)  
 Box [105, 111, 113–115, 118](#)  
 brace [436](#)  
 brackets [156, 157](#)  
 break [324](#)  
 Build [74–76, 87](#)  
 byte [512](#)  
 bytes [439, 514](#)

## C

call [175](#)  
 Calling [336](#)  
 can [121](#)  
 case [155, 193, 433](#)  
 CASE [425](#)  
 CASE, [424](#)  
 Changing [117](#)  
 character [460, 461, 494, 497](#)  
 Character [493](#)  
 clash [511](#)  
 Class [117](#)  
 CLIST [290](#)

- closed [422, 466](#)
  - Closing [436](#)
  - Code Generation Options
    - Associate debug information to assembly source file (-Asmdbg) [95](#)
  - color [204–207](#)
  - COM [109](#)
  - Comma [479, 493, 503](#)
  - command [154, 419](#)
  - Command [107, 108, 119, 121](#)
  - comment [433](#)
  - Comment [239, 264, 411](#)
  - compatibility [191](#)
  - Compatibility [81, 93, 158, 411, 415](#)
  - Compatibility modes Options
    - Add some additional directives [93](#)
    - Alternate comment rules [93](#)
    - Compat [93](#)
    - Ignore FF character at line start Symbol prefixes [93](#)
    - Operator != means equal (-Compat==) [94](#)
    - Support \$ character in symbols [94](#)
    - Support additional ! symbols [94](#)
    - Support FOR directive [93](#)
    - Symbol prefixes (-Compat=s) [93](#)
  - Complex [499](#)
  - conditional [290, 299](#)
  - Conditional [286, 296, 307, 308, 422, 424](#)
  - configuration [126](#)
  - Configuration [111, 113, 393, 401](#)
  - Configure [79, 89, 172, 174, 190](#)
  - Configure listing file Options
    - Disable all (-Lasmc) [90](#)
    - Do not write the absolute line (-Lasmc=a) [90](#)
    - Do not write the address (-Lasmc=l) [90](#)
    - Do not write the include mark column (-Lasmc=i) [90](#)
    - Do not write the location kind (-Lasmc=k) [90](#)
    - Do not write the macro mark (-Lasmc=m) [90](#)
    - Do not write the object code (-Lasmc=c) [90](#)
    - Do not write the relative line (-Lasmc=r) [90](#)
    - Do not write the source line (-Lasmc=s) [90](#)
  - Configuring [48, 108](#)
  - constant [432](#)
  - Constant [230, 292, 294, 458](#)
  - Constant-Definition [283](#)
  - constants [266, 267, 326, 453](#)
  - Constants [266, 411](#)
  - containing [371, 373](#)
  - contains [508](#)
  - context [496](#)
  - continuation [128](#)
  - Continuation [420](#)
  - Control [285, 286](#)
  - Copyright [131](#)
  - COPYRIGHT: [131](#)
  - Core [520](#)
  - Corresponding [384](#)
  - Counter [288, 323](#)
  - Create [30, 202, 226, 321](#)
  - Created [70](#)
  - Creating [37, 69](#)
  - Creation [136](#)
  - current [132](#)
  - Current [124](#)
  - CurrentCommandLine [404](#)
  - currently [488](#)
  - Cut [203](#)
  - Cutting [432, 456, 496](#)
- ## D
- data [164, 508](#)
  - Data [230, 508](#)
  - Data-Allocation [284](#)
  - DCB [294](#)
  - DDE [108](#)
  - debug [155, 194](#)
  - Debug [145](#)
  - declaration [361, 509](#)
  - Declare [328](#)
  - declared [442, 474](#)
  - deep. [426](#)
  - Default [130, 132](#)
  - DEFAULT [424, 425](#)
  - DefaultDir [394](#)
  - DEFAULTDIR: [132](#)
  - Define [162, 292, 294, 295](#)
  - defined [435, 436, 441, 488](#)
  - Defining [234, 235, 335, 369, 372, 387](#)
  - definition [177, 300, 314, 359, 361, 465, 490, 491, 508](#)
  - Definition [332](#)
  - Deleted [39](#)
  - Derivative [161](#)
  - derivatives [521](#)
  - descriptions [287](#)
  - Detailed [287](#)
  - details [129](#)
  - detected [424, 460, 461](#)
  - developers [236](#)
  - development [237](#)
  - different [507](#)
  - differs [509](#)
  - direct [386, 387](#)
  - Direct [250, 261, 333](#)
  - directive [415, 422, 437, 459, 466, 493, 494, 508, 519](#)
  - Directive [248, 283, 491](#)
  - Directive, [460](#)
  - directives [164, 283–287, 416](#)
  - Directives [283, 412](#)
  - directly [488](#)
  - Directly [69, 365](#)
  - directory [124, 132, 139, 185](#)

Directory [29](#)  
 direct-to-direct [255](#)  
 direct-to-indexed [257](#)  
 disable [223](#)  
 Disable [86, 97, 220, 320, 321](#)  
 Display [192](#)  
 division [269](#)  
 Division [427](#)  
 Document [25](#)  
 does [422](#)

## E

Early [237](#)  
 Editor [29, 39, 105–108, 121](#)  
 Editor\_Exe [397, 402](#)  
 Editor\_Name [397, 401](#)  
 Editor\_Opts [398, 402](#)  
 EditorCommandLine [408](#)  
 EditorDDEClientName [408](#)  
 EditorDDEServiceName [409](#)  
 EditorDDETopicName [408](#)  
 EditorType [407](#)  
 ELF [438](#)  
 ELF/DWARF [194](#)  
 else [424](#)  
 ELSE [296](#)  
 Embedded [490](#)  
 Emit [155](#)  
 Enable [192, 310](#)  
 enabled [429](#)  
 end [431, 460, 466, 479](#)  
 End [297–300, 499](#)  
 END [297](#)  
 ENDFOR [298, 505, 506](#)  
 ENDIF [299](#)  
 ENDM [300, 464](#)  
 ENDSW [424](#)  
 Enhanced [237](#)  
 Entries [393, 401](#)  
 entry [131, 287](#)  
 Entry [120](#)  
 environment [165, 195](#)  
 Environment [113, 123, 125, 129, 132, 422](#)  
 ENVIRONMENT: [132](#)  
 EQU [300, 452](#)  
 Equal [504](#)  
 EQUAL [432](#)  
 Equate [300](#)  
 ERRORFILE: [133](#)  
 EVEN [301](#)  
 every [185](#)  
 Example [108, 117, 279, 357, 384, 385, 398, 409](#)  
 Example: [234, 235](#)  
 exceeds [472](#)  
 exist [422](#)  
 expansion [180, 340, 422](#)

Expansion [315](#)  
 expansions [317](#)  
 expected [428, 429, 433, 436, 445, 460, 468–471, 479, 481, 499, 503–505, 517, 518](#)  
 Expected [493](#)  
 expected: [460](#)  
 exported [387, 438, 489](#)  
 Exported [486](#)  
 expression [278, 279, 427, 430, 431, 499](#)  
 Expression [277, 472](#)  
 Extended [251, 262](#)  
 extension [438](#)  
 external [386](#)  
 External [29, 265, 332, 333](#)

## F

FAIL [302, 482, 484](#)  
 failure [458](#)  
 family [161](#)  
 fatal [205](#)  
 feature [185, 186](#)  
 Feedback [120](#)  
 field [240, 362, 363](#)  
 Field [264, 442](#)  
 field: [248](#)  
 Field: [259](#)  
 fields [360](#)  
 file: [438](#)  
 filename [196](#)  
 Filename [133](#)  
 filenames [203](#)  
 files [143–145, 194, 365, 472](#)  
 Files [35, 40–42, 46, 52, 54, 62, 143, 145](#)  
 fixup [455, 456, 458](#)  
 floating [187, 432](#)  
 Floating [433](#)  
 Floating-Point [267](#)  
 force [388](#)  
 Force [276, 301](#)  
 Forcing [313](#)  
 format [166, 203, 208, 210, 212, 213, 215, 217, 433, 515](#)  
 Forward [485](#)  
 found [418, 469, 482, 484](#)  
 Found [509](#)  
 function [356](#)

## G

General [86, 98](#)  
 General Options
 

- MCUasm compatibility (-MCUasm) [98](#)
- MMU Support (-MMU) [98](#)
- Other Flags [98](#)

 Generate [169, 302](#)  
 generated [488](#)



generating [365](#), [454](#)  
 Generating [40](#), [69](#), [74](#), [365](#)  
 Generation [83](#), [95](#)  
 GENPATH: [135](#)  
 Global [105](#), [126](#), [393](#)  
 Graphical [99](#)  
 Group [37](#), [38](#), [394](#)  
 grouping [156](#), [157](#), [338](#)  
 Groups [35](#), [36](#)

## H

has [119](#)  
 HC(S)08/RS08 [25](#)  
 HC08 [241](#), [519](#)  
 HCS08 [76](#), [77](#), [79–81](#), [83](#), [84](#), [86](#), [245](#)  
 header [344](#)  
 Help [168](#)  
 Help) [191](#)  
 here [424](#)  
 HIGH [273](#), [451](#)  
 HIGH\_6\_13 [274](#)  
 Highlights [25](#)  
 Host [83](#), [94](#)  
 Host Options  
   Application Standard Occurrence [95](#)  
   Borrow license feature (-LicBorrow) [95](#)  
   Set environment variable (-Env) [95](#)  
   Wait until a license is available from floating  
   license server (-LicWait) [95](#)  
 How [369](#)

## I

IDE [30](#), [46](#), [60](#), [74](#)  
 identifiers [459](#)  
 IFcc [308](#)  
 ignored [477](#), [495](#), [510](#)  
 ignored. [419](#)  
 ignoring [509](#)  
 Ignoring [494](#)  
 illegal [464](#), [492](#)  
 Illegal [432](#), [434](#), [466](#), [474](#), [478](#), [493–495](#), [497](#), [511](#),  
[512](#), [515](#), [516](#)  
 Immediate [249](#), [263](#)  
 immediate-to-direct [255](#)  
 Implicit [433](#)  
 Improves [164](#)  
 include [384](#), [472](#)  
 Include [143](#), [169](#), [310](#)  
 INCLUDE [310](#)  
 included [182](#)  
 INCLUDETIME: [136](#)  
 increment [256](#), [257](#)  
 index [495](#)  
 Indexed [258](#), [263](#)  
 Indexed, [251–253](#), [258](#)

indexed-to-direct [256](#)  
 Inherent [249](#), [260](#)  
 initialization [126](#)  
 initialized [450–452](#)  
 Initializing [375](#), [378](#), [381](#)  
 input [135](#)  
 Input [52](#), [79](#), [90](#), [119](#), [143](#), [418](#), [498](#)  
 Input options  
   Case insensitivity on label name (-Ci) [91](#)  
   Define label (use spaces to separate labels) (-  
   D) [91](#)  
   Include file search paths (-I) [91](#)  
   Support for structured types (-Struct) [91](#)  
 Insert [324](#), [330](#)  
 instruction [241](#), [246](#), [459](#), [460](#), [466](#), [491](#), [519](#), [521](#)  
 Instruction [241](#), [520](#)  
 instructions [245](#), [520](#)  
 integer [403](#)  
 Integer [266](#)  
 interactive [210](#), [213](#)  
 Interactive [146](#)  
 Interface [99](#)  
 Invalid [460–462](#), [493](#)  
 Invoking [356](#)

## L

label [155](#), [188](#), [434](#), [438](#), [441](#), [460](#), [462](#), [475](#), [508](#),  
[509](#)  
 Label [162](#), [240](#), [437](#), [459](#), [462](#), [477](#), [480](#), [489](#)  
 label: [434](#)  
 label<name> [486](#)  
 labels [452](#)  
 Labels [339](#), [415](#)  
 Language [30](#), [80](#), [81](#), [92](#), [93](#)  
 Language Options  
   Angle brackets for macro arguments grouping  
   (-CMacAngBrack) [92](#)  
   Maximum MacroNest nesting (-MacroNest) [92](#)  
   Square braces for macro arguments grouping (-  
   CMacBrackets) [92](#)  
 large [452](#)  
 Left [429](#)  
 Length [312](#), [325](#), [331](#)  
 level [426](#), [496](#)  
 license [186](#), [187](#)  
 License [184](#), [185](#), [187](#)  
 Limitation: [439](#)  
 Limitations [364](#)  
 limits [281](#)  
 line: [461](#)  
 Lines [330](#)  
 link [475](#)  
 linker [375](#)  
 Linker [65](#)  
 List [152](#), [290](#), [317](#)  
 LIST [310](#)

listing [79](#), [145](#), [146](#), [169](#), [172](#), [174](#), [175](#), [177](#), [180](#),  
[182](#), [226](#), [344](#)  
 Listing [40](#), [89](#), [145](#), [310](#), [320](#), [331](#), [343](#)  
 Listing-File [285](#)  
 LLEN [312](#)  
 Loc [348](#)  
 Local [106](#), [126](#), [401](#)  
 located [333](#)  
 Location [288](#), [323](#)  
 log [419](#)  
 Logical [272](#)  
 long [438](#), [496](#), [498](#)  
 LONGEVEN [313](#)  
 Long-Word [313](#)  
 LOW [274](#), [452](#)

## M

macro [157](#), [177](#), [190](#), [300](#), [314](#), [317](#), [335](#), [422](#), [436](#),  
[460](#), [493](#), [495](#)  
 Macro [156](#), [175](#), [180](#), [248](#), [286](#), [315](#), [335](#), [336](#), [338](#),  
[340](#), [426](#), [463](#), [465–467](#), [488](#)  
 MACRO [314](#)  
 macro) [492](#)  
 macros [125](#), [336](#), [339](#), [341](#)  
 Macros [335](#)  
 Main [100](#)  
 Make [365](#)  
 Management [192](#)  
 Managing [30](#)  
 MAP\_ADDR\_6 [275](#)  
 mapping [419](#)  
 maps [366](#)  
 MASM [411](#)  
 maximum [190](#)  
 Maximum [496](#)  
 mctools.ini [126](#)  
 MCUasm [191](#), [415](#)  
 Memory [189](#), [192](#), [353](#), [366](#)  
 memory, [452](#)  
 Memory-to-memory [255–257](#)  
 Menu [103–105](#)  
 Message/Error [120](#)  
 messages [86](#), [97](#), [201](#), [204–207](#), [218–220](#), [222](#), [418](#)  
 Messages [84](#), [86](#), [95](#), [97](#), [417](#)  
 Messages Options  
   Create err.log Error file [96](#)  
   Create Error Listing File [97](#)  
   Cut file names in Microsoft format to 8.3 (-  
   Wmsg8x3) [96](#)  
   Don't print INFORMATION messages (-W1)  
   [96](#)  
   Don't print INFORMATION or WARNING  
   messages (-W2) [96](#)  
   Maximum number of error messages (-  
   WmsgNe) [97](#)

Messages Options (*index-continued-string*)  
   Maximum number of information messages (-  
   WmsgNi) [97](#)  
   Maximum number of warning messages (-  
   WmsgNw) [97](#)  
   Message Format for batch mode (e.g. %&quot;  
   %f%e%&quot;(%) : %K %d: %m) (-  
   WmsgFob) [97](#)  
   Message Format for no file information (e.g.  
   %K %d: %m) (-WmsgFonf) [97](#)  
   Message Format for no position information  
   (e.g. %&quot;%f%e%&quot;: %K %d: %m) (-  
   WmsgFonp) [97](#)  
   Set message file format for batch mode [96](#)  
   Set messages to Disable [97](#)  
   Set messages to Error [97](#)  
   Set messages to Information [97](#)  
   Set messages to Warning [97](#)

MEXIT [315](#), [492](#)  
 Microsoft [203](#)  
 mismatch [510](#)  
 missing [425](#), [462](#), [463](#), [505](#)  
 Mixed [353](#), [366](#)  
 MLIST [317](#)  
 MMU [521](#)  
 mode [146](#), [208](#), [210](#), [212](#), [213](#), [386](#), [387](#), [452](#), [453](#),  
[520](#)  
 Mode. [512](#)  
 model [189](#)  
 models [353](#)  
 modes [81](#), [93](#), [158](#), [248](#)  
 Modes [259](#)  
 Modifiers [110](#), [150](#)  
 Modularity [236](#)  
 modules [383](#)  
 modulo [269](#)  
 Moving [38](#)  
 multiple [235](#)  
 Multiple [236](#)  
 Multiplication, [269](#)  
 must [459](#), [472](#)

## N

name [436](#), [438](#), [445](#), [460](#), [463](#), [466](#), [468](#), [470](#), [473](#),  
[474](#)  
 Name [140](#)  
 name: [462](#)  
 name>=""> [508](#)  
 names [155](#)  
 Negative [458](#)  
 Nested [341](#)  
 nesting [190](#), [426](#)  
 Nesting [472](#)  
 new [420](#)  
 New [30](#), [37](#), [119](#)  
 NOLIST [320](#)

non-absolute [429](#)  
 NOPAGE [321](#)  
 not [146](#), [195](#), [418](#), [419](#), [422](#), [424](#), [429](#), [433](#), [437](#),  
[442](#), [450–454](#), [457–459](#), [466](#), [469](#), [474](#), [483](#), [485](#),  
[486](#), [490](#), [491](#), [499](#), [515](#), [520](#)  
 Not [488](#), [519](#)  
 notify [192](#)  
 number [289](#), [433](#), [518](#)  
 Number [121](#), [218](#), [219](#), [222](#)

## O

Obj. [350](#)  
 object [131](#), [136](#), [140](#), [365](#), [438](#), [454](#)  
 Object [137](#), [144](#), [196](#), [438](#)  
 Object-code [62](#)  
 objects [429](#)  
 OBJPATH: [137](#)  
 Obsolete [416](#)  
 occurred [417](#), [420](#)  
 occurrence [200](#)  
 offset [251–255](#), [363](#), [458](#)  
 OFFSET [321](#), [489](#)  
 offset, [258](#)  
 Old [420](#)  
 one [234](#), [512](#)  
 Online [191](#)  
 only [520](#), [521](#)  
 only) [126](#)  
 open [419](#), [438](#)  
 operand [515](#)  
 Operand [248](#), [259](#), [515](#)  
 operation [280](#), [432](#)  
 Operation [240](#), [507](#)  
 operations [280](#)  
 operator [273–276](#), [388](#)  
 Operator [274](#), [275](#), [277](#)  
 operators [268–272](#)  
 Operators [268](#), [412](#)  
 option [420](#), [459](#), [520](#)  
 Option [114](#), [152](#), [420](#)  
 options [130](#), [154](#), [429](#)  
 Options [149](#), [407](#)  
 ORG [323](#)  
 out [448](#), [450](#), [455](#), [457](#), [495](#), [519](#)  
 Out [452](#)  
 output [227](#)  
 Output [77](#), [79](#), [88](#), [89](#), [144](#), [166](#)  
 output: [438](#)  
 Output Options  
   Address size in the listing file (-Lasms) [89](#)  
   Do not print included files in listing file (-Li)  
   [89](#)  
   Do not print macro call in listing file (-Lc) [89](#)  
   Do not print macro definition in listing file (-  
   Ld) [89](#)

Output Options (*index-continued-string*)  
   Do not print macro expansion in listing file (-  
   Le) [89](#)  
   Generate listing file ( e.g. %(TEXTPATH)/  
   %n.lst ) (-L) [89](#)  
   -Lasms [89](#)  
   Object File Format (-F) [89](#)  
   Show label statistics (-Li) [89](#)  
 overflow [432](#), [456](#)  
 overflow, [418](#)  
 overlaps [237](#), [456](#)  
 overridden [420](#)  
 overview [335](#)  
 Overview [27](#), [283](#)

## P

Paging [321](#)  
 Panels [75](#), [76](#), [87](#)  
 parameter [488](#)  
 Parameter [354](#)  
 parameters [336](#)  
 parenthesis [428](#), [429](#)  
 passing [354](#)  
 path [130](#), [135](#), [137](#), [139](#), [169](#), [422](#)  
 Path [393](#)  
 PLEN [325](#)  
 point [287](#), [432](#)  
 Point [433](#)  
 Pointer, [254](#), [255](#)  
 portability [237](#)  
 position [217](#)  
 Positive [517](#)  
 possible [457](#)  
 Possible [426](#)  
 post- [256](#), [257](#)  
 post-increment [258](#)  
 precedence [277](#)  
 prefix [495](#)  
 previous [509](#)  
 Previous [496](#)  
 Prints [199](#)  
 PRM [60](#), [375](#), [385](#)  
 processing [147](#)  
 processing. [426](#)  
 Programming [27](#)  
 project.ini) [126](#)  
 Project Information [32](#)  
 projects) [105](#)  
 propagated [458](#)  
 Properties [75](#), [76](#), [87](#)  
 Provide [331](#)

## Q

Qualifier [510](#)

**R**

RAD50 [326, 494](#)  
 RAD50-encoded [326](#)  
 RAM [234, 235, 450–452](#)  
 range [448, 450, 455, 457, 519](#)  
 range: [457](#)  
 reached [496](#)  
 read [154](#)  
 RecentCommandLineX, [403](#)  
 RecentProject0, [396](#)  
 RecentProject1 [396](#)  
 recursion [496](#)  
 recursion? [426](#)  
 Recursive [508](#)  
 redefined [480](#)  
 redefinition [434, 466, 467, 474](#)  
 redefinition: [439](#)  
 reference [485](#)  
 Reference [333](#)  
 referenced [437](#)  
 References [429](#)  
 Rel. [346](#)  
 Relational [272](#)  
 relative [452, 455](#)  
 Relative [254, 262, 453](#)  
 relocatable [279, 372, 373, 378, 499](#)  
 Relocatable [233, 236, 328, 454, 488](#)  
 Removing [38](#)  
 Renaming [38](#)  
 Repeat [305](#)  
 requested [508](#)  
 required [473](#)  
 reserved [434](#)  
 Reserved [266, 459](#)  
 Restoring [39](#)  
 result [280](#)  
 Return [354](#)  
 Reusability [238](#)  
 RGB [204–207](#)  
 Right [428](#)  
 ROM [234, 235](#)  
 RS08 [87–90, 92–95, 97, 98, 246, 520](#)  
 RS08 Assembler Options  
     All options [88](#)  
     Command [88](#)  
     Command line pattern [88](#)  
     Expert Settings [88](#)

**S**

SaveAppearance [395](#)  
 SaveEditor [396](#)  
 SaveOnExit [395](#)  
 SaveOptions [396](#)  
 scheme [354](#)  
 Search [135, 422](#)

SECT [493](#)  
 section [378, 381, 456, 474, 475, 489](#)  
 Section [229, 230, 328, 393–395, 397, 401, 403, 473, 474, 488](#)  
 SECTION [328](#)  
 Section-Definition [283](#)  
 sections [155, 229–231, 233, 236, 369, 371–373, 388](#)  
 Sections [229](#)  
 segmentation [366](#)  
 Select all (-Compat) [93](#)  
 semicolon [460](#)  
 sensitivity [155](#)  
 separate [493](#)  
 Server [187](#)  
 set [241, 246, 455](#)  
 Set [165, 208, 210, 289, 312, 323, 325, 329, 331](#)  
 SET [329, 415, 452, 486](#)  
 Setting [105, 223–225](#)  
 Settings [114, 115](#)  
 Shift [270](#)  
 Short [168, 261](#)  
 SHORT [388](#)  
 Show [188](#)  
 ShowTipOfDay [406](#)  
 Sign [269](#)  
 Simple [279](#)  
 size [174, 439, 452, 456, 494, 509, 510, 516](#)  
 Size [469, 495](#)  
 skipping [418](#)  
 small [476](#)  
 source [155, 355, 356, 369, 372, 378, 381, 466, 497](#)  
 Source [41, 46, 143, 239, 344, 351](#)  
 Source-code [54](#)  
 space [460, 507, 511](#)  
 Space [295](#)  
 SPC [330](#)  
 Special [150, 245](#)  
 specification [132, 133, 196, 469, 494, 516](#)  
 Specify [154, 197](#)  
 Specifying [119](#)  
 Splitting [383](#)  
 Square [157](#)  
 S-Record [138, 145](#)  
 SRECORD: [138](#)  
 Stack [254, 255](#)  
 standard [200, 227](#)  
 start [433](#)  
 Started [107, 108](#)  
 starting [456](#)  
 Starting [99](#)  
 startup [197](#)  
 statistic [419](#)  
 statistics [188](#)  
 Status [102](#)  
 StatusBarEnabled [404](#)  
 stop [508](#)  
 Stop [426](#)  
 string [326](#)

String [267](#), [471](#), [483](#), [495](#)  
 Structure [25](#)  
 structured [198](#), [359](#), [360](#), [362](#)  
 Structured [359](#), [364](#), [439](#)  
 SUBSTR [496](#)  
 subtraction [268](#)  
 Subtraction [507](#)  
 support [164](#)  
 Support [192](#), [198](#), [359](#)  
 supported [433](#), [450–453](#), [486](#), [499](#), [520](#)  
 Switch [155](#), [191](#)  
 SWITCH [424](#)  
 symbol [300](#)  
 Symbol [329](#), [332](#), [333](#), [436](#), [470](#)  
 symbol: [435](#), [436](#)  
 Symbol-Linkage [284](#)  
 symbols [264–266](#), [321](#), [386](#), [387](#)  
 Symbols [264](#), [333](#)  
 Syntax [239](#)

## T

Tab [331](#)  
 table [375](#)  
 Table [378](#), [381](#)  
 TABS [331](#)  
 taken [432](#)  
 Temporary [139](#)  
 Terminate [315](#)  
 terminated [459](#)  
 text [310](#)  
 Text [139](#)  
 TEXTPATH: [139](#)  
 this [25](#), [419](#), [433](#), [475](#), [496](#)  
 This [521](#)  
 time [136](#)  
 Tiny [260](#)  
 TipFilePos [406](#)  
 Title [100](#), [331](#)  
 TITLE [331](#)  
 TMP: [139](#)  
 to... [369](#)  
 too [426](#), [438](#), [452](#), [476](#), [477](#), [486](#), [498](#)  
 Tool [74](#)  
 Toolbar [102](#), [119](#)  
 ToolbarEnabled [404](#)  
 tools [105](#)  
 tools) [106](#)  
 Tracking [237](#)  
 Translation [281](#)  
 truncated [482](#), [512](#), [514](#)  
 two [514](#)  
 type [138](#), [359](#), [360](#), [439](#), [441](#), [442](#), [490](#), [491](#)  
 Type [441](#), [445](#)  
 type: [364](#)  
 types [198](#), [230](#), [359](#)  
 Types [360](#)

## U

Unary [280](#)  
 Undeclared [435](#), [436](#)  
 Undefined [265](#)  
 Unit [192](#)  
 Unknown [417](#)  
 unsupported [493](#)  
 use [195](#)  
 Use [119–121](#)  
 used [419](#)  
 user [86](#), [97](#), [207](#), [220](#), [435](#), [436](#)  
 User [99](#), [140](#), [508](#)  
 USERNAME: [140](#)

## V

value [300](#), [420](#), [511](#), [517](#)  
 Value [329](#), [354](#), [448](#), [450](#), [455](#), [457](#), [476](#), [477](#), [482](#),  
[486](#), [512](#), [514](#), [519](#)  
 Value<value> [457](#)  
 values [507](#)  
 variable [165](#), [362](#)  
 Variable [361](#), [509](#)  
 variables [129](#), [355](#)  
 Vector [375](#), [378](#), [381](#)  
 version [199](#)  
 version. [419](#)  
 very [496](#)  
 View [105](#)  
 vs. [236](#)

## W

Wait [187](#)  
 warning [201](#), [207](#)  
 Warning [222](#), [225](#)  
 was [496](#)  
 when [429](#)  
 will [154](#)  
 window [146](#)  
 Window [100](#), [121](#)  
 WindowFont [405](#)  
 WindowPos [405](#)  
 Wizard [70](#)  
 word [301](#)  
 Working [27](#), [369](#), [372](#)  
 Write [227](#)  
 Writing [41](#)

## X

XDEF [332](#), [509](#)  
 XREF [333](#), [509](#)  
 XREF, [509](#)  
 XREFB [333](#)

## **Y**

your [41](#)

## **Z**

Zero [427](#)



**How to Reach Us:**

**Home Page:**

[freescale.com](http://freescale.com)

**Web Support:**

[freescale.com/support](http://freescale.com/support)

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale 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 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 does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [freescale.com/SalesTermsandConditions](http://freescale.com/SalesTermsandConditions).

Freescale, the Freescale logo, and Processor Expert are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners.

© 2010–2014 Freescale Semiconductor, Inc.