

CodeWarrior Development Studio for Microcontrollers V10.x S12Z Assembler Reference Manual

Document Number: CWMCUS12ZASMREF
Rev 10.6, 02/2014

Contents

Section number	Title	Page
Chapter 1		
Using S12Z Assembler		
1.1	Highlights.....	19
1.2	Structure of this Document.....	19
1.3	Accompanying Documentation.....	20
Chapter 2		
Working with Assembler		
2.1	Overview.....	23
2.1.1	Project Directory.....	24
2.1.2	External Editor.....	25
2.2	Managing Assembly Language Project Using CodeWarrior IDE.....	25
2.2.1	Create New Project.....	25
2.2.2	Additional Project Information.....	27
2.3	Analysis of Groups and Files in a Project.....	30
2.3.1	CodeWarrior Groups.....	31
2.3.2	Creating New Group.....	32
2.3.3	Adding New File to the Project.....	32
2.3.4	Renaming File or Group.....	33
2.3.5	Moving File.....	33
2.3.6	Removing File.....	33
2.3.7	Restoring Deleted File.....	33
2.3.8	Using Editor.....	34
2.3.9	Generating Listing Files.....	35
2.4	Writing your Assembly Source Files.....	36
2.5	Analyzing Project Files.....	37
2.6	Assembling Source Files.....	39
2.6.1	Assembling and Linking with CodeWarrior IDE.....	39

Section number	Title	Page
2.6.2	Assembling with Assembler.....	40
2.6.2.1	Configuring Assembler.....	41
2.6.2.2	Input Files.....	45
2.6.2.3	Assembling Assembly Source-code Files.....	46
2.7	Linking Application.....	53
2.7.1	Linking with CodeWarrior IDE.....	53
2.7.1.1	PRM File.....	53
2.7.1.2	Linking Object-code Files.....	55
2.7.2	Linking with Linker.....	56
2.8	Assembler Build Properties Panels.....	60
2.8.1	S12Z Assembler Build Properties Panels.....	60
2.8.1.1	S12Z Assembler.....	61
2.8.1.2	S12Z Assembler > Output.....	61
2.8.1.3	S12Z Assembler > Output > Configure Listing File.....	62
2.8.1.4	S12Z Assembler > Input.....	63
2.8.1.5	S12Z Assembler > Language.....	63
2.8.1.6	S12Z Assembler > Language > Compatibility Modes.....	63
2.8.1.7	S12Z Assembler > Host.....	64
2.8.1.8	S12Z Assembler > Code Generation.....	64
2.8.1.9	S12Z Assembler > Messages.....	65
2.8.1.10	S12Z Assembler > Messages > Disable User Messages.....	66
2.8.1.11	S12Z Assembler > General.....	66

Chapter 3 Assembler Graphical User Interface

3.1	Starting Assembler.....	69
3.2	Assembler Main Window.....	70
3.2.1	Window Title.....	71
3.2.2	Content Area.....	71
3.2.3	Toolbar.....	72

Section number	Title	Page
3.2.4	Status Bar.....	73
3.2.5	Assembler Menu Bar.....	73
3.2.6	File Menu.....	73
3.2.7	Assembler Menu.....	75
3.2.8	View Menu.....	75
3.3	Editor Setting Dialog Box.....	75
3.3.1	Global Editor (shared by all tools and projects).....	76
3.3.2	Local Editor (shared by all tools).....	76
3.3.3	Editor Started with Command Line.....	77
3.3.3.1	Example of Configuring a Command Line Editor.....	78
3.3.4	Editor Started with DDE.....	78
3.3.5	CodeWarrior with COM.....	79
3.3.6	Modifiers.....	80
3.4	Save Configuration Dialog Box.....	81
3.4.1	Environment Configuration Dialog Box.....	83
3.5	Option Settings Dialog Box.....	84
3.6	Message Settings Dialog Box.....	85
3.6.1	Changing the Class Associated with a Message.....	87
3.6.1.1	Example.....	87
3.7	About Dialog Box.....	88
3.8	Specifying Input File.....	89
3.8.1	Use Command Line in Toolbar to Assemble.....	89
3.8.2	Assembling a New File.....	89
3.8.3	Assembling a File which has Already been Assembled.....	89
3.8.4	Use File > Assemble Entry.....	90
3.8.5	Use Drag and Drop.....	90
3.9	Message/Error Feedback.....	90
3.9.1	Use Information from Assembler Window.....	91
3.9.2	Use User-defined Editor.....	91

Section number	Title	Page
3.9.3	Line Number can be Specified on the Command Line.....	91
3.9.4	Line Number cannot be Specified on the Command Line.....	91

Chapter 4 Environment

4.1	Current directory.....	94
4.2	Environment macros.....	95
4.3	Global initialization file - mctools.ini (PC only).....	96
4.4	Local configuration file (usually project.ini).....	96
4.5	Line continuation.....	98
4.6	Environment variables details.....	99
4.6.1	ABSPATH: Absolute file path.....	99
4.6.2	ASMOPTIONS: Default assembler options.....	100
4.6.3	COPYRIGHT: Copyright entry in object file.....	101
4.6.4	DEFAULTDIR: Default current directory.....	102
4.6.5	ENVIRONMENT: Environment file specification.....	102
4.6.6	ERRORFILE: Filename specification error.....	103
4.6.7	GENPATH: Search path for input file.....	105
4.6.8	INCLUDETIME: Creation time in the object file.....	106
4.6.9	OBJPATH: Object file path.....	107
4.6.10	SRECORD: S-Record type.....	108
4.6.11	TEXTPATH: Text file path.....	109
4.6.12	TMP: Temporary directory.....	109
4.6.13	USERNAME: User Name in object file.....	110

Chapter 5 Files

5.1	Input files.....	113
5.1.1	Source files.....	113
5.1.2	Include files.....	113

Section number	Title	Page
5.2	Output files.....	114
5.2.1	Object files.....	114
5.2.2	Absolute files.....	114
5.2.3	S-Record Files.....	115
5.2.4	Listing files.....	115
5.2.5	Debug listing files.....	115
5.2.6	Error listing file.....	116
5.2.6.1	Interactive mode (Assembler window open).....	116
5.2.6.2	Batch mode (Assembler window not open).....	116
5.3	File processing.....	117

Chapter 6 Assembler Options

6.1	Types of Assembler Options.....	119
6.2	Assembler Option Details.....	121
6.2.1	Using special modifiers.....	121
6.2.1.1	Examples using special modifiers.....	122
6.3	List of Assembler Options.....	124
6.4	Detailed Listing of All Assembler Options.....	126
6.4.1	-ArgFile: Specify a file from which additional command line options will be read.....	126
6.4.2	-AsmDbg: Emit assembly source file information in debug sections.....	127
6.4.3	-Ci: Switch case sensitivity on label names OFF.....	127
6.4.4	-CMacAngBrack: Angle brackets for grouping Macro Arguments.....	128
6.4.5	-CMacBrackets: Square brackets for macro arguments grouping.....	129
6.4.6	-Compat: Compatibility modes.....	130
6.4.7	-D: Define Label.....	133
6.4.8	-DefLabel: Improves support for data allocation directives.....	135
6.4.9	-Env: Set environment variable.....	136
6.4.10	-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format.....	137
6.4.11	-H: Short Help.....	138

Section number	Title	Page
6.4.12	-I: Include file path.....	139
6.4.13	-L: Generate a listing file.....	140
6.4.14	-Lasmc: Configure listing file.....	142
6.4.15	-Lasms: Configure the address size in the listing file.....	144
6.4.16	-Lc: No Macro call in listing file.....	146
6.4.17	-Ld: No macro definition in listing file.....	148
6.4.18	-Le: No Macro expansion in listing file.....	150
6.4.19	-Li: No included file in listing file.....	152
6.4.20	-Lic: License information.....	154
6.4.21	-LicA: License information about every feature in directory.....	155
6.4.22	-LicBorrow: Borrow license feature.....	156
6.4.23	-LicWait: Wait until floating license is available from floating License Server.....	158
6.4.24	-Ll: Show label statistics.....	159
6.4.25	-MacroNest: Configure maximum macro nesting.....	159
6.4.26	Message A1004 (available in the Online Help).....	160
6.4.27	-MCUasm: Switch compatibility with MCUasm ON.....	160
6.4.28	-N: Display notify box.....	161
6.4.29	-NoBeep: No beep in case of an error.....	162
6.4.30	-NoDebugInfo: No debug information for ELF/DWARF files.....	163
6.4.31	-NoEnv: Do not use environment.....	163
6.4.32	-ObjN: Object filename specification.....	164
6.4.33	-Prod: Specify project file at startup.....	165
6.4.34	-Struct: Support for structured types.....	166
6.4.35	-V: Prints the Assembler version.....	167
6.4.36	-View: Application standard occurrence.....	168
6.4.37	-W1: No information messages.....	169
6.4.38	-W2: No information and warning messages.....	170
6.4.39	-WErriFile: Create "err.log" error file.....	170
6.4.40	-Wmsg8x3: Cut filenames in Microsoft format to 8.3.....	172

Section number	Title	Page
6.4.41	-WmsgCE: RGB color for error messages.....	173
6.4.42	-WmsgCF: RGB color for fatal messages.....	173
6.4.43	-WmsgCI: RGB color for information messages.....	174
6.4.44	-WmsgCU: RGB color for user messages.....	175
6.4.45	-WmsgCW: RGB color for warning messages.....	176
6.4.46	-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode.....	177
6.4.47	-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode.....	178
6.4.48	-WmsgFob: Message format for batch mode.....	180
6.4.49	-WmsgFoi: Message format for interactive mode.....	182
6.4.50	-WmsgFonf: Message format for no file information.....	184
6.4.51	-WmsgFonp: Message format for no position information.....	185
6.4.52	-WmsgNe: Number of error messages.....	187
6.4.53	-WmsgNi: Number of Information messages.....	188
6.4.54	-WmsgNu: Disable user messages.....	189
6.4.55	-WmsgNw: Number of Warning messages.....	190
6.4.56	-WmsgSd: Setting a message to disable.....	191
6.4.57	-WmsgSe: Setting a message to Error.....	192
6.4.58	-WmsgSi: Setting a message to Information.....	193
6.4.59	-WmsgSw: Setting a Message to Warning.....	193
6.4.60	-WOutFile: Create error listing file.....	194
6.4.61	-WStdout: Write to standard output.....	195

Chapter 7 Sections

7.1	Section attributes.....	197
7.1.1	Code sections.....	197
7.1.2	Constant sections.....	198
7.1.3	Data sections.....	198
7.2	Section types.....	198
7.2.1	Absolute sections.....	199

Section number	Title	Page
7.2.2	Relocatable sections.....	201
7.2.2.1	Example: Defining one RAM and one ROM area.....	202
7.2.2.2	Example: Defining multiple RAM and ROM areas.....	203
7.3	Relocatable vs. absolute sections.....	204
7.3.1	Modularity.....	204
7.3.2	Multiple developers.....	204
7.3.3	Early development.....	205
7.3.4	Enhanced portability.....	205
7.3.5	Tracking overlaps.....	206
7.3.6	Reusability.....	206

Chapter 8 Assembler Syntax

8.1	Comment line.....	207
8.2	Source line.....	207
8.2.1	Label field.....	208
8.2.2	Operation field.....	209
8.2.2.1	Instruction set.....	209
8.2.2.1.1	HCS12Z Instruction Set.....	209
8.2.2.2	Directive.....	216
8.2.2.3	Macro.....	217
8.2.3	Operand Field: Addressing Modes.....	217
8.2.3.1	Inherent Addressing Mode.....	218
8.2.3.2	Register Addressing Mode.....	218
8.2.3.3	Immediate Addressing Mode.....	218
8.2.3.4	Short Immediate Addressing Mode (IMMe4).....	219
8.2.3.5	Relative Addressing Modes (REL, REL1).....	219
8.2.3.6	Extended Addressing Modes (EXT1, EXT2, EXT3, EXT24).....	219
8.2.3.7	Indexed Addressing Modes	220
8.2.3.7.1	4-Bit Short Constant Offset from X, Y, or SP (IDX).....	220

Section number	Title	Page
8.2.3.7.2	9-Bit Constant Offset from X, Y, SP or PC (IDX1).....	221
8.2.3.7.3	24-Bit Constant Offset from X, Y, SP or PC (IDX3).....	221
8.2.3.7.4	Register Offset Indexed from X, Y, or SP (REG,IDX).....	222
8.2.3.7.5	Automatic Pre/Post Increment/Decrement from X, Y, or SP (++IDX).....	222
8.2.3.7.6	18-Bit Constant Offset from Di (IDX2,REG).....	223
8.2.3.7.7	24-Bit Constant Offset from Di (IDX3,REG).....	223
8.2.3.8	Indexed Indirect Addressing Modes	223
8.2.3.8.1	Register Offset Indexed Indirect from X or Y ([REG,IDX]).....	223
8.2.3.8.2	9-Bit Constant Offset Indexed Indirect from X, Y, SP or PC ([IDX1]).....	224
8.2.3.8.3	24-Bit Constant Offset Indexed Indirect from X, Y, SP or PC ([IDX3]).....	224
8.2.3.8.4	Address Indirect Addressing Mode ([EXT3]).....	224
8.3	Symbols.....	225
8.3.1	User-defined symbols.....	225
8.3.2	External symbols.....	226
8.3.3	Undefined symbols.....	226
8.3.4	Reserved symbols.....	227
8.4	Constants.....	227
8.4.1	Integer constants.....	228
8.4.2	String constants.....	228
8.4.3	Floating-Point constants.....	229
8.5	Operators.....	229
8.5.1	Addition and subtraction operators (binary).....	229
8.5.2	Multiplication, division and modulo operators (binary).....	230
8.5.3	Sign operators (unary).....	231
8.5.4	Shift operators (binary).....	231
8.5.5	Bitwise operators (binary).....	232
8.5.6	Bitwise operators (unary).....	233
8.5.7	Logical operators (unary).....	233
8.5.8	Relational operators (binary).....	234

Section number	Title	Page
8.5.9	HIGH operator.....	235
8.5.10	HIGH_6_13 Operator.....	236
8.5.11	LOW operator.....	236
8.5.12	MAP_ADDR_6 Operator.....	237
8.5.13	Force operator (unary).....	237
8.5.14	Operator precedence.....	238
8.6	Expression.....	239
8.6.1	Absolute expression.....	240
8.6.1.1	Example.....	240
8.6.2	Simple relocatable expression.....	241
8.6.3	Unary operation result.....	241
8.6.4	Binary operations result.....	242
8.7	Translation limits.....	242

Chapter 9 Assembler Directives

9.1	Directive Overview.....	245
9.1.1	Section-Definition Directives.....	245
9.1.2	Constant-Definition directives.....	246
9.1.3	Data-Allocation directives.....	246
9.1.4	Symbol-Linkage directives.....	246
9.1.5	Assembly-Control directives.....	247
9.1.6	Listing-File Control directives.....	247
9.1.7	Macro Control directives.....	248
9.1.8	Conditional Assembly directives.....	248
9.2	Detailed descriptions of all assembler directives.....	249
9.2.1	ABSENTRY - Application entry point.....	249
9.2.2	ALIGN - Align Location Counter.....	250
9.2.3	BASE - Set number base.....	251
9.2.4	CLIST - List conditional assembly.....	252

Section number	Title	Page
9.2.5	DC - Define Constant.....	254
9.2.6	DCB - Define Constant Block.....	255
9.2.7	DS - Define Space.....	257
9.2.8	ELSE - Conditional assembly.....	258
9.2.9	END - End assembly.....	260
9.2.10	ENDFOR - End of FOR block.....	261
9.2.11	ENDIF - End conditional assembly.....	261
9.2.12	ENDM - End macro definition.....	262
9.2.13	EQU - Equate symbol value.....	263
9.2.14	EVEN - Force word alignment.....	263
9.2.15	FAIL - Generate Error message.....	264
9.2.16	FOR - Repeat assembly block.....	267
9.2.17	IF - Conditional assembly.....	269
9.2.18	IFcc - Conditional assembly.....	271
9.2.19	INCLUDE - Include text from another file.....	272
9.2.20	LIST - Enable Listing.....	273
9.2.21	LLEN - Set Line Length.....	274
9.2.22	LONGEVEN - Forcing Long-Word alignment.....	275
9.2.23	MACRO - Begin macro definition.....	276
9.2.24	MEXIT - Terminate Macro Expansion.....	277
9.2.25	MLIST - List macro expansions.....	279
9.2.26	NOLIST - Disable Listing.....	282
9.2.27	NOPAGE - Disable Paging.....	283
9.2.28	OFFSET - Create absolute symbols.....	284
9.2.29	ORG - Set Location Counter.....	286
9.2.30	PAGE - Insert Page break.....	287
9.2.31	PLEN - Set Page Length.....	288
9.2.32	RAD50 - RAD50-encoded string constants.....	288
9.2.33	SECTION - Declare Relocatable Section.....	290

Section number	Title	Page
9.2.34	SET - Set Symbol Value.....	292
9.2.35	SPC - Insert Blank Lines.....	293
9.2.36	TABS - Set Tab Length.....	293
9.2.37	TITLE - Provide Listing Title.....	294
9.2.38	XDEF - External Symbol Definition.....	294
9.2.39	XREF - External Symbol Reference.....	295
9.2.40	XREFB - External Reference for Symbols located on the Direct Page.....	296

Chapter 10 Macros

10.1	Macro Overview.....	297
10.2	Defining Macro.....	297
10.3	Calling Macros.....	298
10.4	Macro Parameters.....	298
10.4.1	Macro Argument Grouping.....	299
10.5	Labels Inside Macros.....	301
10.6	Macro Expansion.....	302
10.7	Nested Macros.....	302

Chapter 11 Assembler Listing File

11.1	Page Header.....	306
11.2	Source Listing.....	306
11.2.1	Abs.....	306
11.2.2	Rel.....	307
11.2.3	Loc.....	308
11.2.4	Obj. code.....	309
11.2.5	Source Line.....	309

Chapter 12 Mixed C and Assembler Applications

12.1	Parameter Passing Scheme.....	311
12.2	Return Value.....	311

Section number	Title	Page
12.3	Accessing Assembly Variables in ANSI-C Source File.....	311
12.4	Accessing ANSI-C Variables in Assembly Source File.....	312
12.5	Invoking Assembly Function in ANSI-C Source File.....	313
12.5.1	Example of a C File.....	314
12.6	Support for Structured Types.....	316
12.6.1	Structured Type Definition.....	316
12.6.2	Types Allowed for Structured Type Fields.....	317
12.6.3	Variable Definition.....	318
12.6.4	Variable Declaration.....	318
12.6.5	Accessing Structured Variable.....	319
12.6.5.1	Accessing Field Address.....	319
12.6.5.2	Accessing Field Offset.....	320
12.7	Structured Type: Limitations.....	321

Chapter 13 Make Applications

13.1	Assembly Applications.....	323
13.1.1	Directly Generating Absolute File.....	323
13.1.1.1	Generating Object Files.....	323
13.1.2	Mixed C and Assembly Applications.....	324
13.2	Memory Maps and Segmentation.....	324

Chapter 14 How to...

14.1	Working with Absolute Sections.....	327
14.1.1	Defining Absolute Sections in Assembly Source File.....	327
14.1.2	Linking Application Containing Absolute Sections.....	329
14.2	Working with Relocatable Sections.....	330
14.2.1	Defining Relocatable Sections in Source File.....	330
14.2.2	Linking Application Containing Relocatable Sections.....	331

Section number	Title	Page
14.3	Initializing Vector Table.....	333
14.3.1	Initializing the Vector table in the linker PRM file.....	333
14.3.2	Initializing Vector Table in Source File Using Relocatable Section.....	336
14.3.3	Initializing Vector Table in Source File Using Absolute Section.....	339
14.4	Splitting Application into Modules.....	341
14.4.1	Example of Assembly File (Test1.asm).....	342
14.4.2	Corresponding Include File (Test1.inc).....	342
14.4.3	Example of Assembly File (Test2.asm).....	343
14.4.3.1	Example of PRM File (Test2.prm).....	343

**Chapter 15
Appendices**

**Chapter 16
Symbols and Notation**

**Chapter 17
Global Configuration File Entries**

17.1	[Installation] Section.....	351
17.1.1	Path.....	351
17.1.2	Group.....	352
17.2	[Options] Section.....	352
17.2.1	DefaultDir.....	352
17.3	[XXX_Assembler] Section.....	353
17.3.1	SaveOnExit.....	353
17.3.2	SaveAppearance.....	353
17.3.3	SaveEditor.....	354
17.3.4	SaveOptions.....	354
17.3.5	RecentProject0, RecentProject1.....	354
17.4	[Editor] Section.....	355
17.4.1	Editor_Name.....	355
17.4.2	Editor_Exe.....	356
17.4.3	Editor_Opts.....	356

Section number	Title	Page
17.5	Example.....	357

Chapter 18 Local Configuration File Entries

18.1	[Editor] Section.....	359
18.1.1	Editor_Name.....	359
18.1.2	Editor_Exe.....	360
18.1.3	Editor_Opts.....	360
18.2	[XXX_Assembler] Section.....	361
18.2.1	RecentCommandLineX, X= integer.....	362
18.2.2	CurrentCommandLine.....	362
18.2.3	StatusbarEnabled.....	362
18.2.4	ToolbarEnabled.....	363
18.2.5	WindowPos.....	363
18.2.6	WindowFont.....	364
18.2.7	TipFilePos.....	364
18.2.8	ShowTipOfDay.....	365
18.2.9	Options.....	365
18.2.10	EditorType.....	365
18.2.11	EditorCommandLine.....	366
18.2.12	EditorDDEClientName.....	366
18.2.13	EditorDDETopicName.....	367
18.2.14	EditorDDEServiceName.....	367
18.3	Example.....	368

Chapter 19 MASM Compatibility

19.1	Comment Line.....	369
19.2	Constants (Integers).....	369
19.3	Operators.....	370
19.3.1	Directives.....	370

Section number	Title	Page
Chapter 20 MCUasm Compatibility		
20.1	Labels.....	373
20.2	SET directive.....	373
20.3	Obsolete directives.....	374

Chapter 1

Using S12Z Assembler

This document explains how to effectively use the S12Z Macro Assembler.

1.1 Highlights

The major features of the S12Z 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.

- [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.
- [Symbols and Notation](#) : Description of the expressions of instruction that represents the variable content.

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.

1.3 Accompanying Documentation

The **Documentation** page describes the documentation included in the *CodeWarrior Development Studio for Microcontrollers v10.x*. You can access the **Documentation** by:

- opening the `START_HERE.html` in `<CWInstallDir>\MCU\Help` folder,
- selecting **Help > Documentation** from the IDE's menu bar, or selecting the **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > Documentation** from the Windows taskbar.

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and

debugger, refer to the *CodeWarrior Common Features Guide* in this folder: `<CWInstallDir>\MCU\Help\PDF`



Chapter 2

Working with Assembler

This chapter is primarily a tutorial for creating and managing S12Z 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.

In this chapter:

- [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](#)
- [Assembler Build Properties Panels](#)

2.1 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. See the [Mixed C and Assembler Applications](#) chapter.

This manual covers the Assembler dedicated to the Freescale S12Z series of microcontrollers. The S12Z 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>\s12lisa_support` 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:

```
<CWInstallDir>\MCU\prog
```

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.

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 **S12Z > S12ZVH Family > MC9S12ZVH64**.
7. Click **Next**.

The **Connections** page appears.

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

The **Language and Build Tools Options** page appears.

10. Select the **ASM** option and select the floating point format and the memory model appropriate for your project. By default **None** option is selected for best code density.
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 S12Z 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.

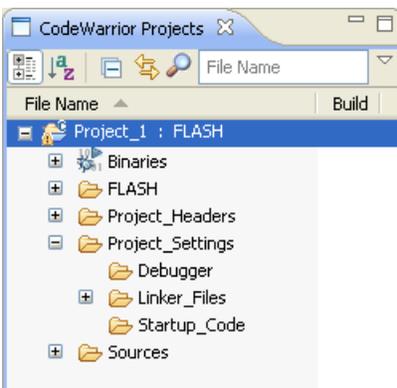


Figure 2-1. CodeWarrior Projects View

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.

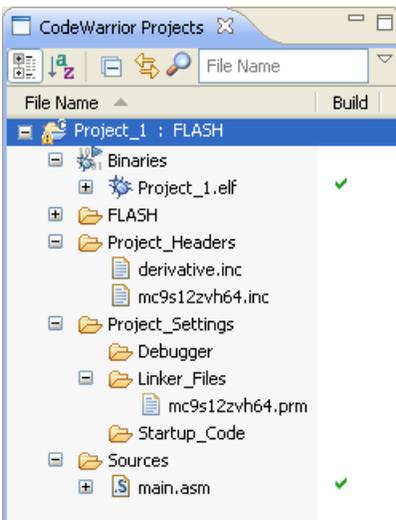


Figure 2-2. CodeWarrior Projects View Showing Files

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 (`.elf`) 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, launch configuration , and debug configuration file.
 - The `Linker_Files` folder stores the linker command file (`.prm`).
 - The `Startup_Code` folder has a C file that initializes the Microcontrollers's stack and critical registers when the program launches.
- `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.

```

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 the      *
;* Freescale CodeWarrior for the S12Z Program directory *
;*****
; Include derivative-specific definitions
    INCLUDE 'derivative.inc'

; export symbols
    XDEF Entry, _Startup, main
    ; we use export 'Entry' as symbol. This allows us to
    ; reference 'Entry' 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

; variable/data section
MY_EXTENDED_RAM: SECTION
; Insert here your data definition.
Counter      ds.w 1
FiboRes      ds.w 1

; code section
MyCode:      SECTION
main:
_Startup:
Entry:
    LD S, #__SEG_END_SSTACK - 1 ; initialize the stack pointer
    CLI                          ; enable interrupts
    
```

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 linker command file (`.prm`).

- Startup Code

This group contains the source code that manages the Microcontrollers initialization and startup functions. For S12Z derivatives, these functions appear in the source file `start.c`.

- Project_Headers

This group holds include files. One include file is for the particular CPU derivative. In this case, the `mc9s12zvh64.inc` file is for the `MC9S12ZVH64` 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. A dialog box appears asking you 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.

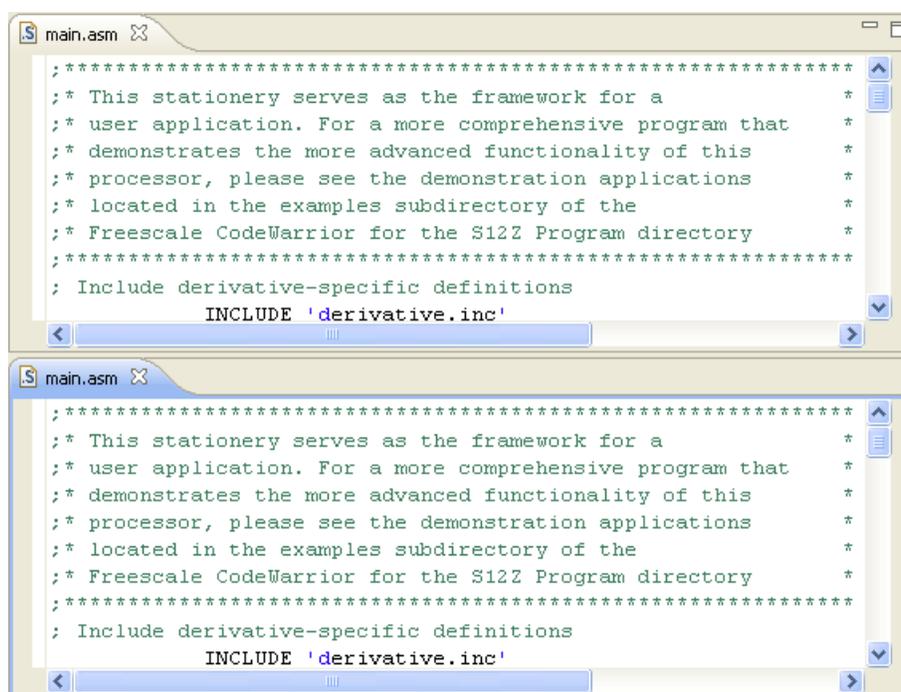


Figure 2-4. Editor Area Showing Tiled Source Files

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 **S12Z Assembler > Output** in the **Tool Settings** page.
5. Specifies the name, %n, and path, %TEXTSPATH, of the assembly listing file in the **Generate listing file (e.g. %(TEXTSPATH)/%n.lst)** text box. For example, enter `<Workspace>\Project_1\lis.lst` in the text box, if `<Workspace>\Project_1` and `lis.lst` are the path and name of the listing file, where `<Workspace>` is the current working directory.
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.

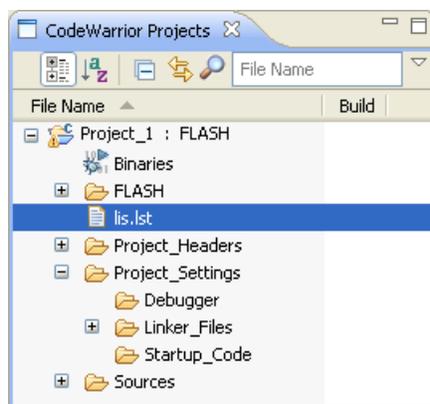


Figure 2-5. CodeWarrior Projects View - Assembly Listing File

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.

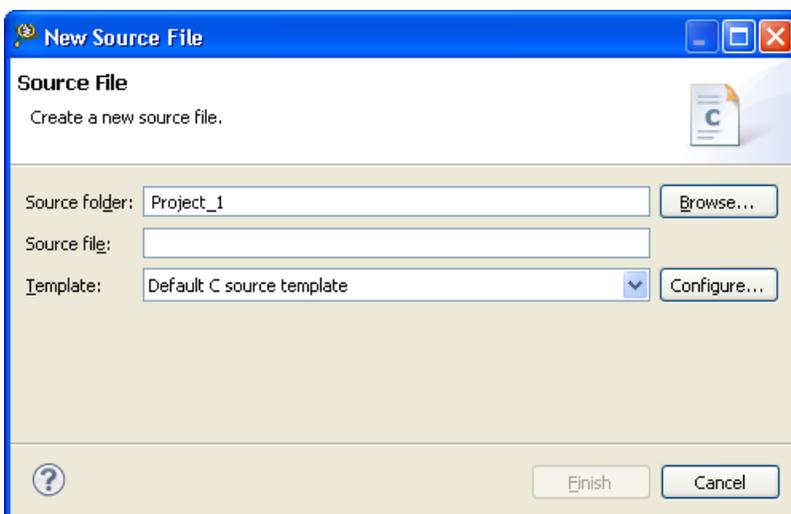


Figure 2-6. New Source File Dialog Box

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 the S12Z Program directory  *
;*****
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
    
```

```

; export symbols

        XDEF Entry, _Startup, main
        ; we use export 'Entry' as symbol. This allows us to
        ; reference 'Entry' 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

; variable/data section
MY_EXTENDED_RAM: SECTION

; Insert here your data definition.
Counter      ds.w 1
FiboRes      ds.w 1

; code section
MyCode:      SECTION

main:
_Startup:
Entry:

        LD S, #__SEG_END_SSTACK - 1    ; initialize the stack pointer
        CLI                          ; enable interrupts

EndlessLoop:

        LD D2, #1                      ; D2 contains counter

CouterLoop:

        ST D2, Counter                 ; update global.
        BSR CalcFibo
        ST D4, FiboRes                 ; store result
        LD D2, Counter
        INC D2
        CMP D2, #24                    ; larger values cause overflow.
        BNE CouterLoop
        BRA EndlessLoop               ; restart.

; Function to calculate fibonacci numbers. Argument is in D2.
CalcFibo:

        LD D3, #0                      ; second last
        LD D4, #0                      ; last
        DBEQ D2, FiboDone              ; loop once more (if D2 was 1,

```

```
were done already)

FiboLoop:

        ADD D3, D4                ; overwrite second last with new
value
        EXG D3, D4                ; exchange them -> order is
correct again
        DBNE D2,FiboLoop

FiboDone:

        RTS                       ; result in D4
```

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`.

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>\S121isa_Tools` 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 >* .elf

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 S12Z 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 `as121isa.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.

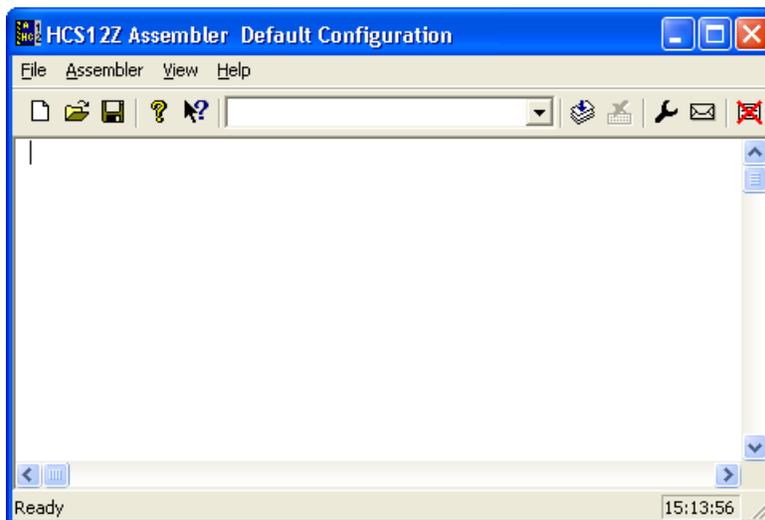


Figure 2-7. S12Z Assembler Default Configuration Dialog Box

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>\<MCU>\S121isa_Tools` 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.

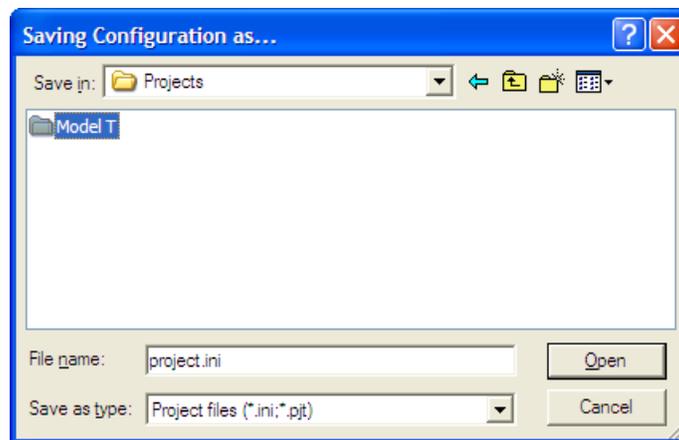


Figure 2-8. Loading Configuration Dialog Box

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.

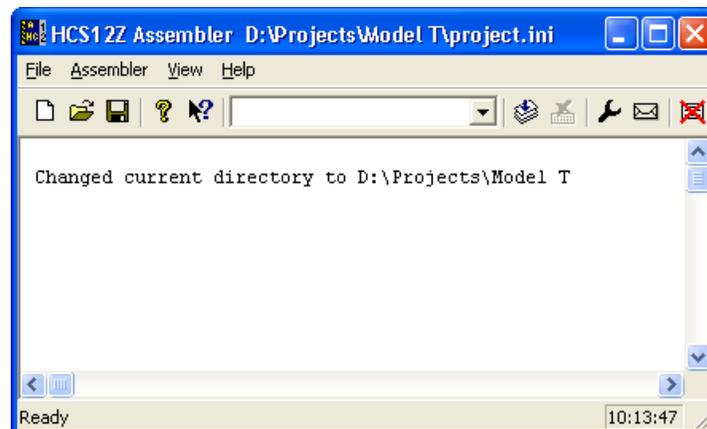


Figure 2-9. Assembler Displaying Current Directory

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.

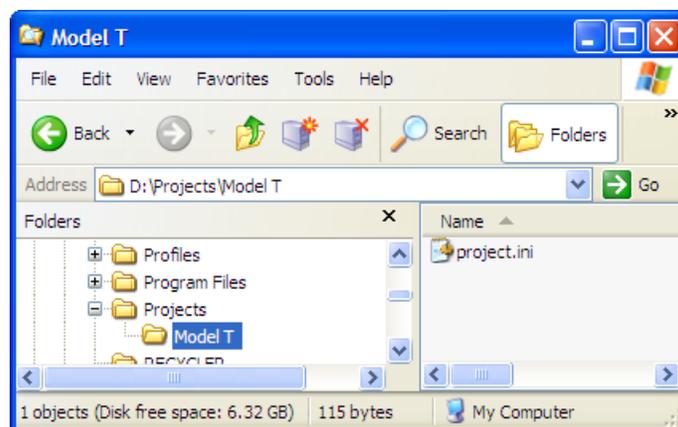


Figure 2-10. Project directory in Windows Explorer

If you further examined the contents of the `project.ini` configuration file, you would see that it contains Assembler options in the `[AHCS12Z_Assembler]` portion of the file. The `project.ini` file for this project only has an `[AHCS12Z_Assembler]` section, as the following listing displays.

Listing: Contents of project.ini File

```
[AHCS12Z_Assembler]
StatusBarEnabled=1

ToolbarEnabled=1

WindowPos=0,1,-1,-1,-1,-1,494,406,953,697

EditorType=4
```

The `AHCS12Z_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 **HCS12Z Assembler Option Settings** dialog box 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 options displayed 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.

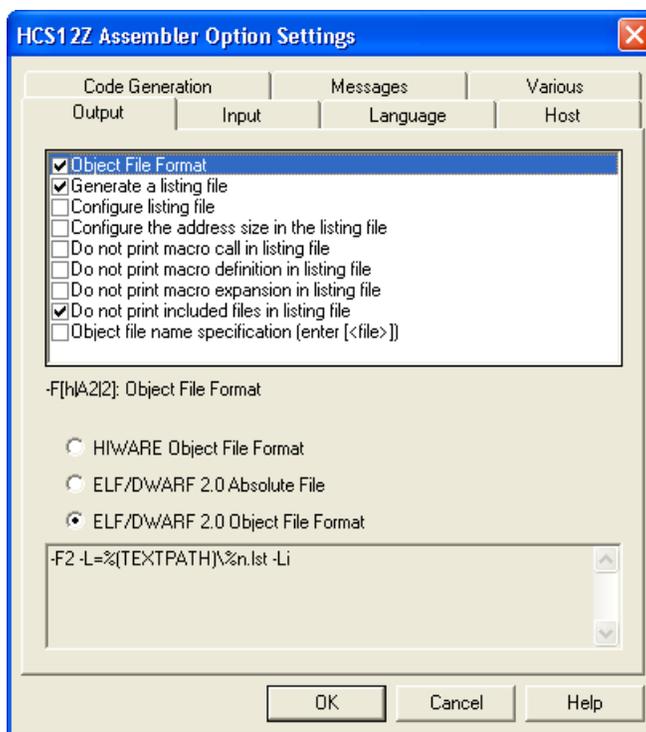


Figure 2-11. S12Z Assembler Option Settings Dialog Box

5. Click **OK** to close the **HCS12Z 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.

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

```
[AHCS12Z_Assembler]
StatusBarEnabled=1

ToolbarEnabled=1

WindowPos=0,1,-32000,-32000,-1,-1,495,471,941,739

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`, `derivative.inc` and the `mc9s12zvh64.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 [main.asm file](#) shows.

Now there are four files in the project (refer to the figure listed below):

- the `project.ini` configuration file and
- two files in the `Sources` folder:
 - `main.asm`
 - `derivative.inc`
 - `mc9s12zvh64.inc`

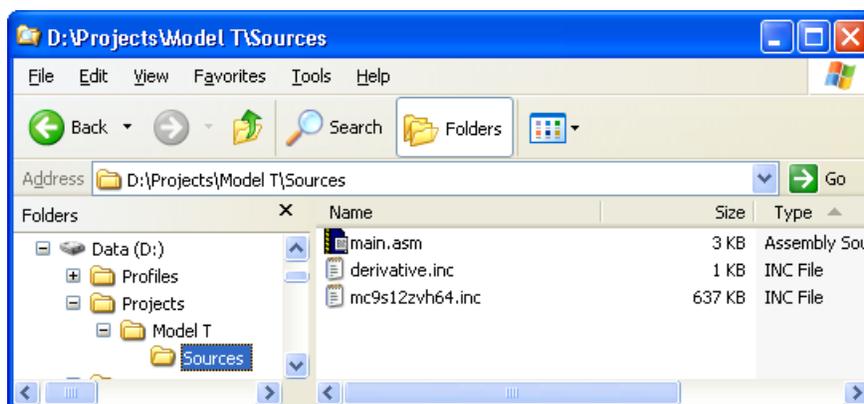


Figure 2-12. Project Files

2.6.2.3 Assembling Assembly Source-code Files

To 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.

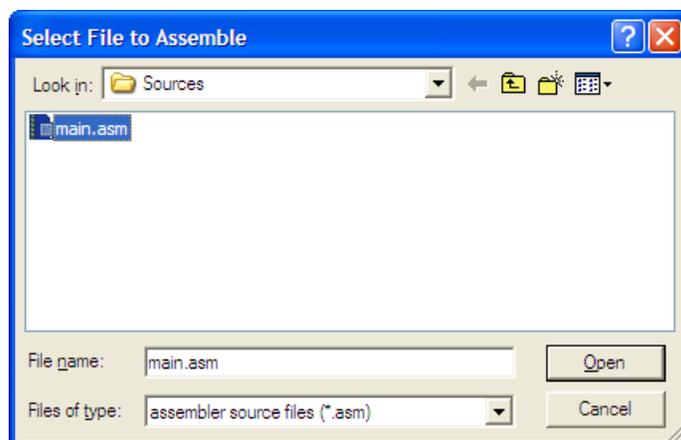


Figure 2-13. Select File to Assemble Dialog Box

3. Click **Open**.

The `main.asm` file starts assembling.

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

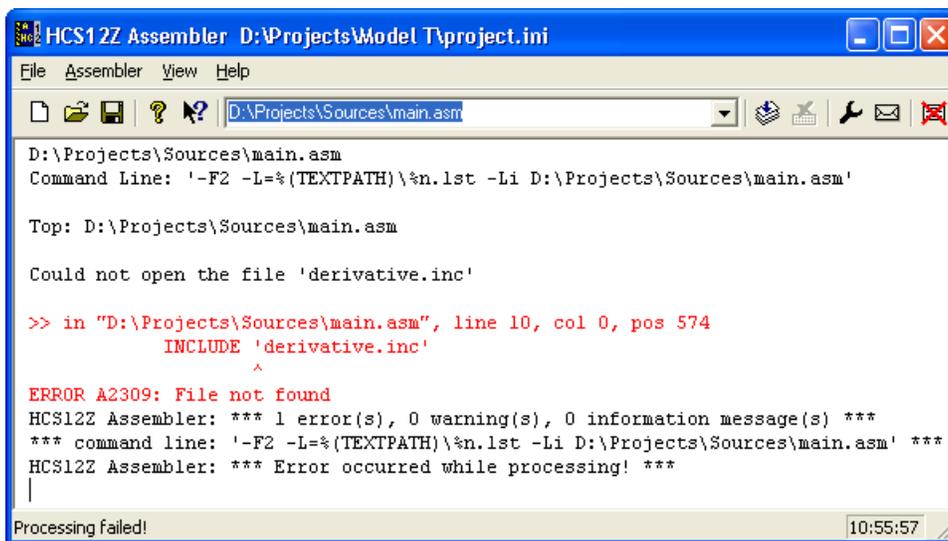


Figure 2-14. Results of Assembling main.asm File

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 [main.asm file](#).

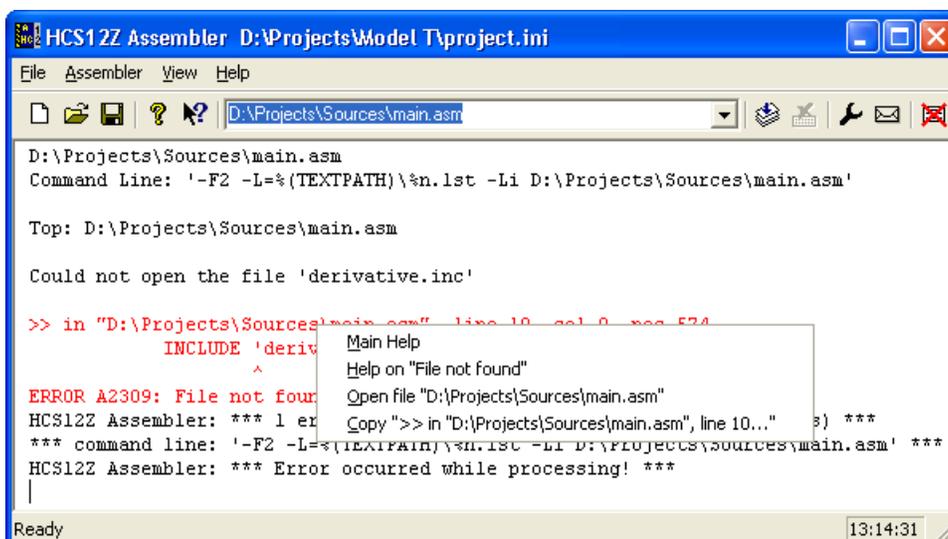


Figure 2-15. Context Menu

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

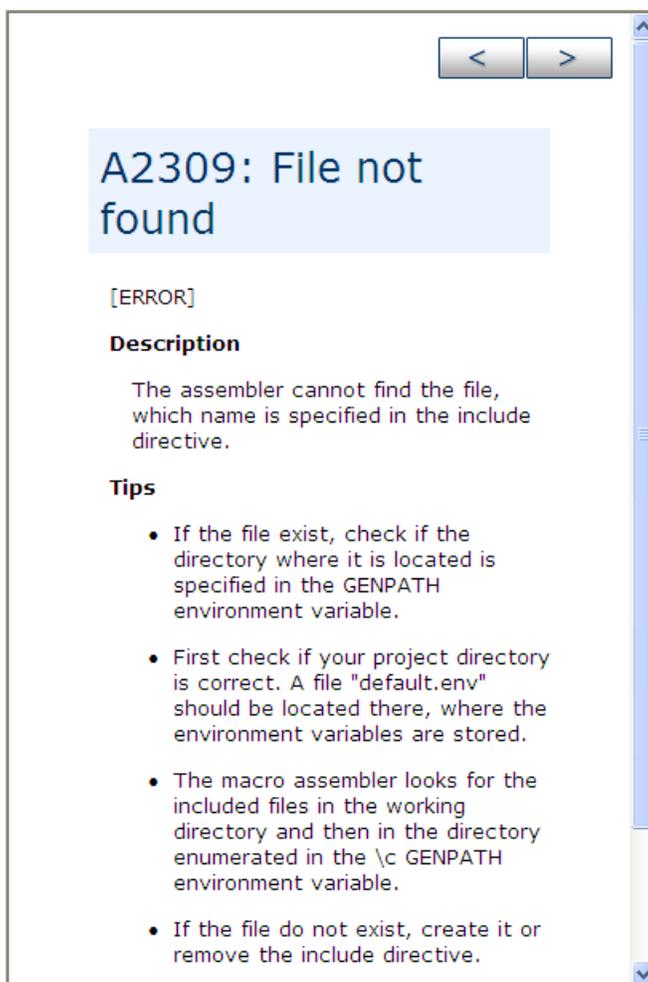


Figure 2-16. A2309: File not found

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. Click the **Environment** tab and then select **General Path**.
3. Click 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.

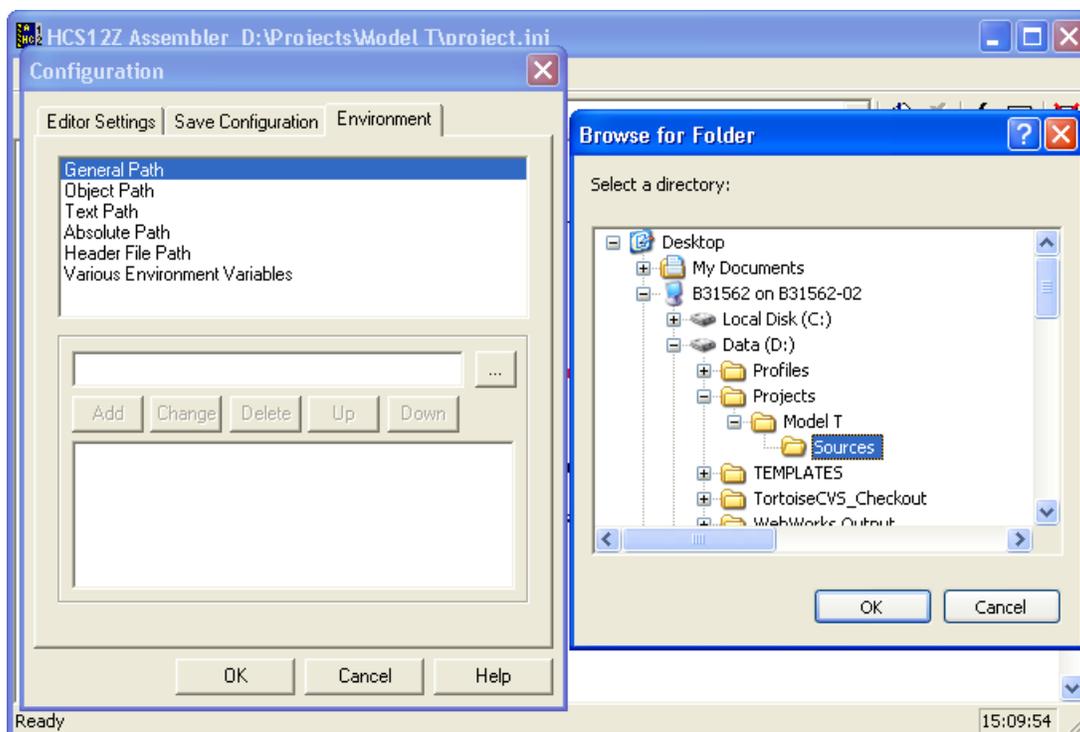


Figure 2-17. Browsing for Sources Folder

4. Click **OK** to close the **Browse for Folder** dialog box.
5. The **Configuration** dialog box is active again. Click the **Add** button.

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

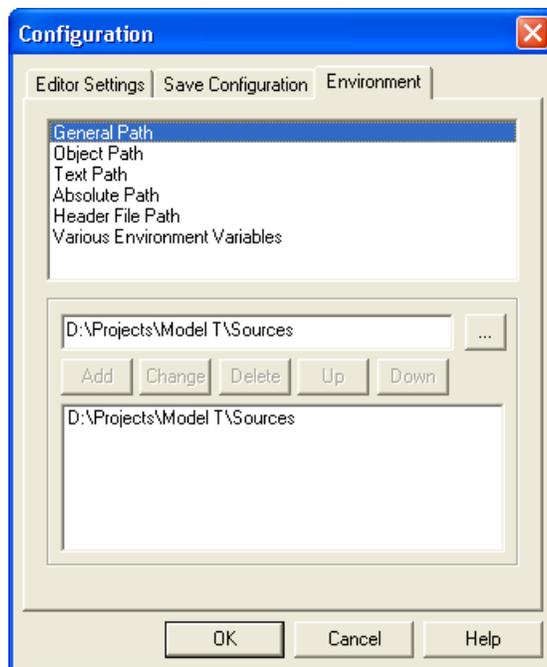


Figure 2-18. Adding GENPATH

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 for 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**. The following image displays the successful assembling of the project with the second attempt.

```

HCS12Z Assembler D:\Projects\Model T\project.ini
File Assembler View Help
"D:\Projects\Model T\Sources\main.asm"
Command Line: '-F2 -L=%(TEXTSPATH)\%n.lst -Li "D:\Projects\Model T\Sources\main.asm"'

Top: D:\Projects\Model T\Sources\main.asm

"D:\Projects\Model T\Sources\derivative.inc"
"D:\Projects\Model T\Sources\mc9s12zvh64.inc"
writing debug listing to D:\Projects\Model T\main.dbg
Output file: "D:\Projects\Model T\main.o"
Code Size: 52
writing listing to D:\Projects\Model T\main.lst
HCS12Z Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HCS12Z Assembler: *** Processing ok ***

Processing ok 10:36:40
    
```

Figure 2-19. Assemble Attempt #2 - Success!

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.

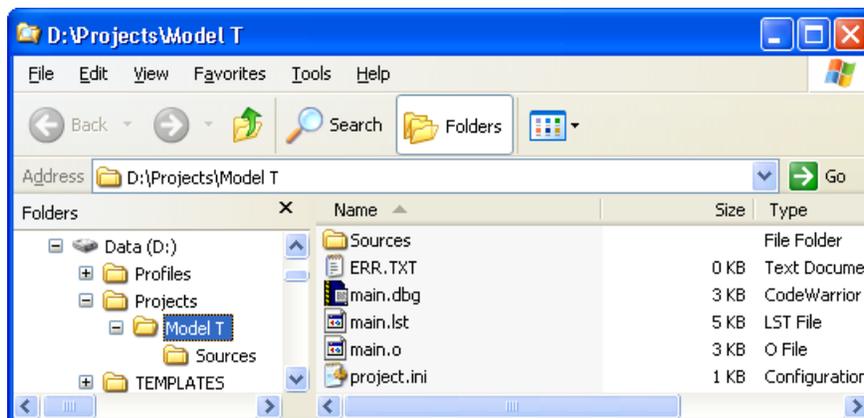


Figure 2-20. Project Directory After Successful Assembly

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

```
[AHCS12Z_Assembler]
StatusBarEnabled=1

ToolBarEnabled=1

WindowPos=0,1,-1,-1,-1,-1,319,392,953,747

EditorType=4

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

RecentCommandLine0="D:\Projects\Model T\Sources\main.asm"
CurrentCommandLine="D:\Projects\Model T\Sources\main.asm"
RecentCommandLine1=D:\Projects\Sources\main.asm

[Environment Variables]
GENPATH=D:\Projects\Model T\Sources

OBJPATH=

TEXTSPATH=

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 MC9S12ZVH64 derivative - <derivative>.prm

```
/* This is a linker parameter file for the MC9S12ZVH64 */
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. */

/* Register space */

/
* IO_SEG = PAGED 0x000000 TO 0x000FFF; intentionally
not defined */

/* RAM */

RAM = READ_WRITE 0x001000 TO 0x001FFF;

/* EEPROM */
```

Linking Application

```

EEPROM    = READ_ONLY    0x100000    TO    0x100FFF;

/* non-paged FLASHs */

ROM       = READ_ONLY    0xFF0000    TO    0xFFFFDFF;

/* VECTORS = READ_ONLY    0xFFFFE00    TO    0xFFFFFFFF; intentionally
not defined: used for VECTOR commands below */

//OSVECTORS = READ_ONLY    0xFFFFE10    TO    0xFFFFFFFF; /* OSEK
interrupt vectors (use your vector.o) */

END

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

    _PRESTART,           /* Used in HIWARE format: jump to
_Startup at the code start */

    STARTUP,             /* startup data structures */

    ROM_VAR,             /* constant variables */

    STRINGS,             /* string literals */

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

    //.ostext,           /* OSEK */

    NON_BANKED,         /* runtime routines which must not be
banked */

    DEFAULT_ROM,

    COPY                INTO    ROM;

    //.stackstart,       /* eventually used for OSEK kernel
awareness: Main-Stack Start */

    SSTACK,             /* allocate stack first to avoid
overwriting variables on overflow */

    //.stackend,         /* eventually used for OSEK kernel
awareness: Main-Stack End */

    DEFAULT_RAM         INTO    RAM;

    //.vectors           INTO    OSVECTORS; /* OSEK */

END

ENTRIES /* keep the following unreferenced variables */

    /* OSEK: always allocate the vector table and all dependent
objects */

    //_vectab OsBuildNumber _OsOrtiStackStart _OsOrtiStart

END

STACKSIZE 0x100

VECTOR 0 _Startup /* reset vector: this is the default entry point for
a C/C++ application. */

//VECTOR 0 Entry /* reset vector: this is the default entry point for
an Assembly application. */

```

```
//INIT Entry      /* for assembly applications: that this is as well  
the initialization entry point */
```

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 `<derivative>.prm` file in [Linker PRM file for MC9S12ZVH64 derivative - <derivative>.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 `*.elf` and a `*.map` file in the `FLASH` subfolder in the project directory.

The `Project_1.elf` and `Project_1.map` files 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.

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>.prm file, change the name of the executable (*.elf) file to whatever you choose, for example, <project_name>.elf. 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 MC9S12ZVH64 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 MC9S12ZVH64 */
LINK Model_T.elf /* 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,
    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> \S121isa_Tools` 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.

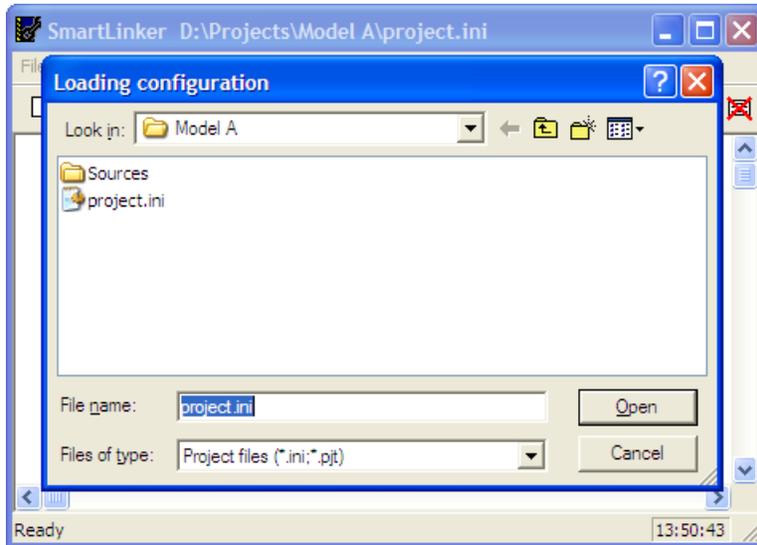


Figure 2-21. Microcontroller Linker

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 figure listed below).
11. Browse to locate and select the PRM file for your project.

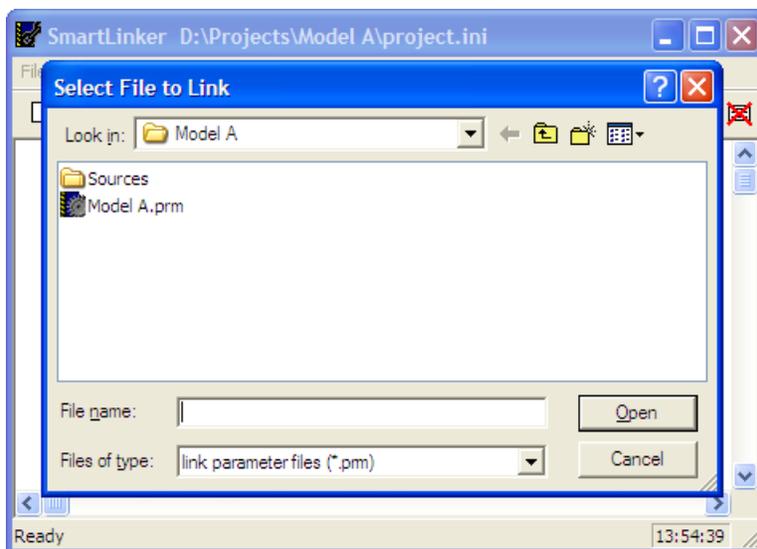


Figure 2-22. Select File to Link Dialog Box

12. Click **Open**.

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

The following image displays the smart linker window after linking.

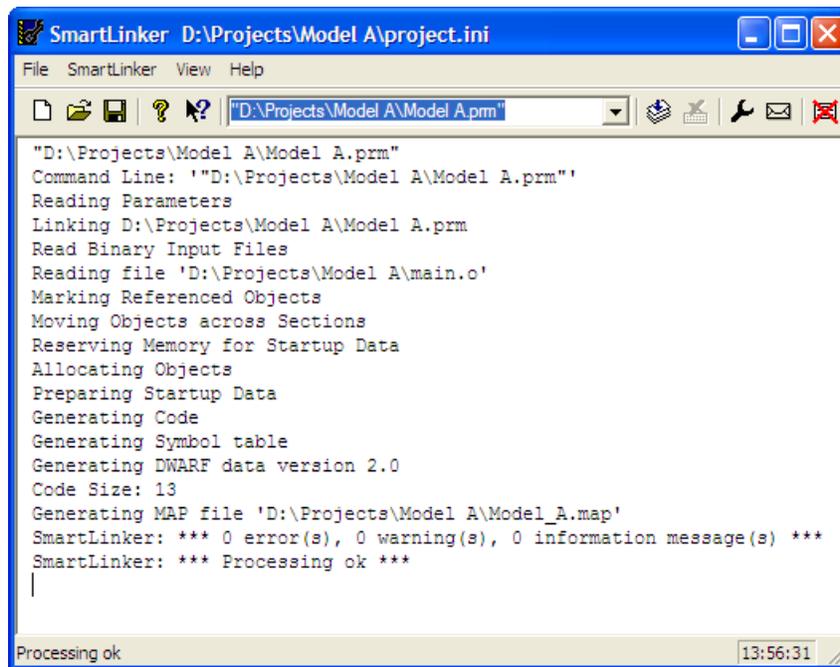


Figure 2-23. SmartLinker Window After Linking

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 `*.elf` 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.

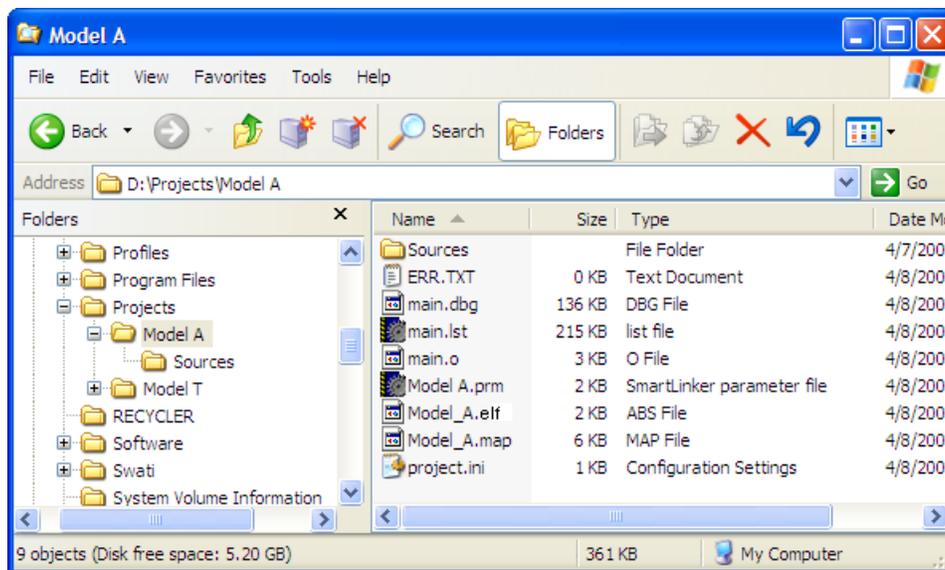


Figure 2-24. Project Directory After Linking

2.8 Assembler Build Properties Panels

The following sections describe how to configure the [S12Z 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.

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.

2.8.1 S12Z Assembler Build Properties Panels

The following listed are the build properties panels for the S12Z Assembler.

NOTE

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

Table 2-1. Build Properties Panel for S12Z Assembler

Build Tool	Build Properties Panels
S12Z Assembler	S12Z Assembler > Output
	S12Z Assembler > Output > Configure Listing File
	S12Z Assembler > Input
	S12Z Assembler > Language
	S12Z Assembler > Language > Compatibility Modes
	S12Z Assembler > Host
	S12Z Assembler > Code Generation
	S12Z Assembler > Messages
	S12Z Assembler > Messages > Disable User Messages
	S12Z Assembler > General

2.8.1.1 S12Z Assembler

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

The following table lists and describes the assembler options for S12Z architectures.

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 command line pattern; default is <code>\${COMMAND} \${FLAGS} \${OUTPUT_FLAG}\${OUTPUT_PREFIX}\${OUTPUT} \${INPUTS}</code>

2.8.1.2 S12Z 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 table lists and describes the output options for S12Z assembler.

Table 2-3. Tool Settings - S12Z Assembler > Output Options

Option	Description
Object File Format	For description, refer to the topic -F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format .
Show Label Statistics	For description, refer to the topic -LI: Show label statistics .
Generate Listing File (e.g. % (TEXTPATH) /%n .lst)	For description, refer to the topic -L: Generate a listing file .
Address Size in the Listing File (integer)	For description, refer to the topic -Lasms: Configure the address size in the listing file .
Do Not Print Macro Call in Listing File	For description, refer to the topic -Lc: No Macro call in listing file .
Do Not Print Macro Definition in Listing File	For description, refer to the topic -Ld: No macro definition in listing file .
Do Not Print Macro Expansion in Listing File	For description, refer to the topic -Le: No Macro expansion in listing file .
Do Not Print Included Files in Listing File	For description, refer to the topic -Li: No included file in listing file .

2.8.1.3 S12Z Assembler > Output > Configure Listing File

Use this panel to specify the general assembler behavior. The following image shows the **Configure Listing File** panel options.



Figure 2-25. Tool Settings - S12Z Assembler > Output > Configure Listing File

NOTE

For information about the description of the options available in the **Configure Listing File** panel, refer to the topic [-Lasmc: Configure listing file](#).

2.8.1.4 S12Z Assembler > Input

Use this panel to specify file search paths and any additional include files the **S12Z Assembler** should use. You can specify multiple search paths and the order in which you want to perform the search.

The following table lists and describes the input options for S12Z assembler.

Table 2-4. Tool Settings - S12Z Assembler > Input Options

Option	Description
Case Insensitivity on Label Name	For description, refer to the topic -Ci: Switch case sensitivity on label names OFF .
Define Label (use spaces to separate labels)	For description, refer to the topic -D: Define Label .
Support for Structured Types	For description, refer to the topic -Struct: Support for structured types .
Include File Search Path	For description, refer to the topic -I: Include file path .

2.8.1.5 S12Z Assembler > Language

Use this panel to specify code- and symbol-generation options for the S12Z assembler.

The following table lists and describes the language options for S12Z assembler.

Table 2-5. Tool Settings - S12Z Assembler > Language Options

Option	Description
Angle Brackets for Macro Arguments Grouping	For description, refer to the topic -CMacAngBrack: Angle brackets for grouping Macro Arguments .
Square Braces for Macro Arguments Grouping	For description, refer to the topic -CMacBrackets: Square brackets for macro arguments grouping .
Maximum Macro Nesting	For description, refer to the topic -MacroNest: Configure maximum macro nesting .

2.8.1.6 S12Z Assembler > Language > Compatibility Modes

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

- Select All
- Symbol Prefixes
- Ignore FF Character at Line Start
- Alternate Comment Rules
- Support FOR Directive
- Support Additional Directives
- Operator != Means Equal
- Support \$ Character in Symbols
- Support Additional ! Operators

Figure 2-26. Tool Settings - S12Z Assembler > Language > Compatibility Modes

NOTE

For information about the description of the options available in the **Compatibility Modes** panel, refer to the topic [-Compat: Compatibility modes](#).

2.8.1.7 S12Z Assembler > Host

Use this panel to specify the host settings of the S12Z assembler.

The following table lists and describes the memory model options for S12Z assembler.

Table 2-6. Tool Settings - S12Z Assembler > Host Options

Option	Description
Borrow License Feature	For description, refer to the topic -LicBorrow: Borrow license feature .
Wait Until a License is Available from Floating License Server	For description, refer to the topic -LicWait: Wait until floating license is available from floating License Server .
Application Standard Occurrence	For description, refer to the topic -View: Application standard occurrence .
Set Environment Variable	For description, refer to the topic -Env: Set environment variable .

2.8.1.8 S12Z Assembler > Code Generation

Use this panel to specify the code generation assembler behavior.

The following table lists and describes the code generation assembler options for S12Z assembler.

Table 2-7. Tool Settings - S12Z Assembler > Code Generation Options

Option	Description
Associate Debug Information to Assembly Source File	For description, refer to the topic -AsmDbg: Emit assembly source file information in debug sections.

2.8.1.9 S12Z Assembler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target.

The following table lists and describes the messages options.

Table 2-8. Tool Settings - S12Z Assembler > Messages Options

Option	Description
Don't Print INFORMATION Messages	For description, refer to the topic -W1: No information messages.
Don't Print INFORMATION or WARNING Messages	For description, refer to the topic -W2: No information and warning messages.
Create err.log Error File	For description, refer to the topic -WErrFile: Create "err.log" error file.
Create Error Listing File	For description, refer to the topic -WOutFile: Create error listing file.
Cut File Names in Microsoft Format to 8.3	For description, refer to the topic -Wmsg8x3: Cut filenames in Microsoft format to 8.3.
Set Message File Format for Batch Mode	For description, refer to the topic -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode.
Message Format for Batch Mode (e.g. %"%f%e%" (%l) : %K %d: %m\n)	For description, refer to the topic -WmsgFob: Message format for batch mode.
Message Format for No File Info (e.g. %K %d: %m\n)	For description, refer to the topic -WmsgFonf: Message format for no file information.
Message Format for No Position Info (e.g. %"%f%e%" : %K %d: %m\n)	For description, refer to the topic -WmsgFonp: Message format for no position information.
Maximum Number of Error Messages	For description, refer to the topic -WmsgNe: Number of error messages.
Maximum Number of Information Messages	For description, refer to the topic -WmsgNi: Number of Information messages.
Set Messages to Disable	For description, refer to the topic -WmsgSd: Setting a message to disable.

Table continues on the next page...

Table 2-8. Tool Settings - S12Z Assembler > Messages Options (continued)

Option	Description
Set Messages to Error	For description, refer to the topic -WmsgSe: Setting a message to Error .
Set Messages to Warning	For description, refer to the topic -WmsgSw: Setting a Message to Warning .
Set Messages to Information	For description, refer to the topic -WmsgSi: Setting a message to Information .

2.8.1.10 S12Z Assembler > Messages > Disable User Messages

Use this panel to specify the options for disabling the user messages for the S12Z assembler. The following image shows the **Disable User Messages** panel options.

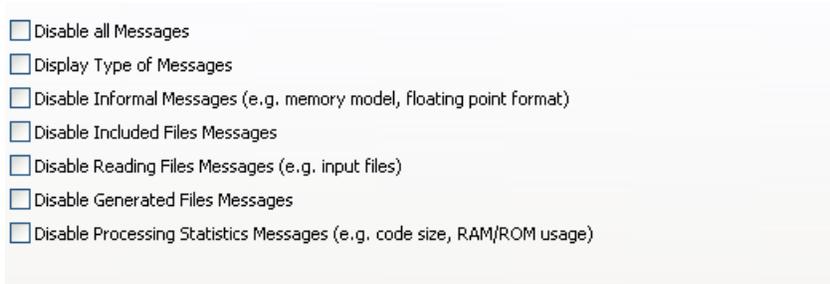


Figure 2-27. Tool Settings - S12Z Assembler > Messages > Disable User Messages

NOTE

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

2.8.1.11 S12Z Assembler > General

Use this panel to specify the general assembler behavior.

The following table lists and describes the general assembler options for S12Z.

Table 2-9. Tool Settings - S12Z Assembler > General Options

Option	Description
MCUasm Compatibility	For description, refer to the topic -MCUasm: Switch compatibility with MCUasm ON .

Table continues on the next page...

Table 2-9. Tool Settings - S12Z Assembler > General Options (continued)

Option	Description
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, as the following image displays.

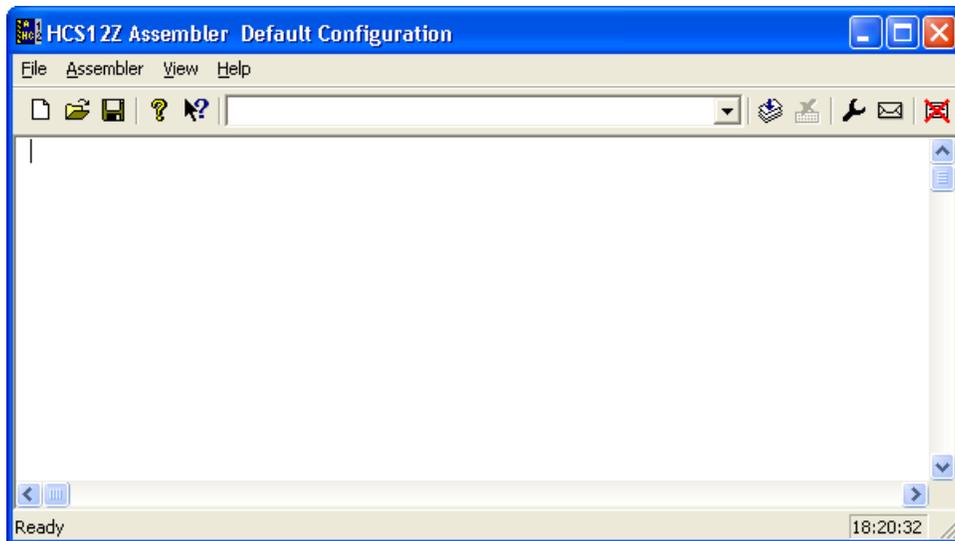


Figure 3-2. HCS12Z Assembler Main Window

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 3-3. 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 Open File 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 Open File dialog box. The selected file is assembled when the Open File dialog box is closed by clicking OK .
New/Default Configuration	Resets the Assembler option settings to their default values. The default Assembler options which are activated are specified in the Assembler Options chapter.
Load Configuration	A standard Open File 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 Open File 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 Save As 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 Save As dialog box. The current settings are saved in the specified configuration file when the Save As dialog box is closed by clicking OK .
Configuration...	Opens the Configuration dialog box to specify the editor used for error feedback and which parts to save with a configuration. See Editor Setting Dialog Box and Save Configuration Dialog Box .
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 Option Settings Dialog Box).
Messages	Maps messages to a different message class (see Message Settings Dialog Box).
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 figure 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.

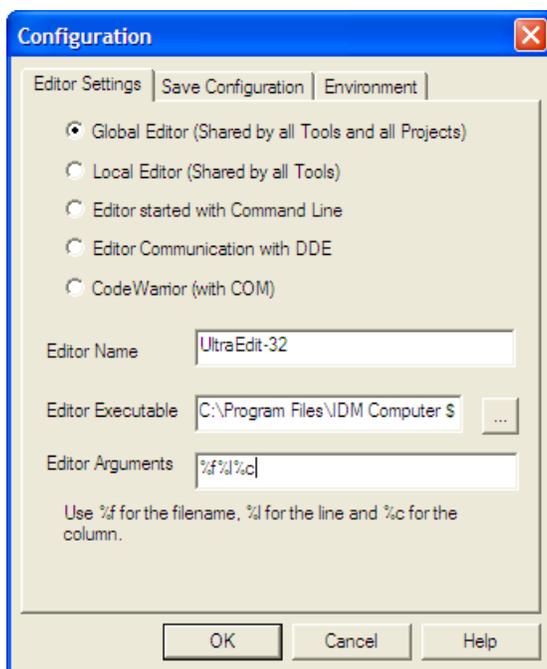


Figure 3-4. Global Editor Configuration Dialog Box

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.

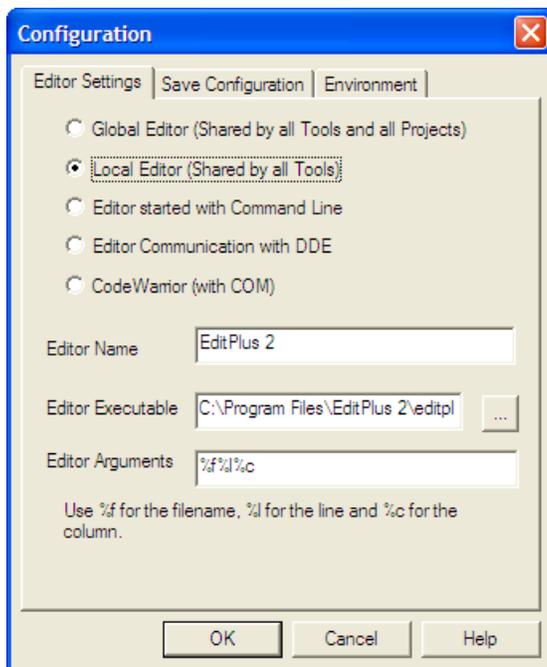


Figure 3-5. Local Editor Configuration Dialog Box

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. (See the [Modifiers](#) section below.)

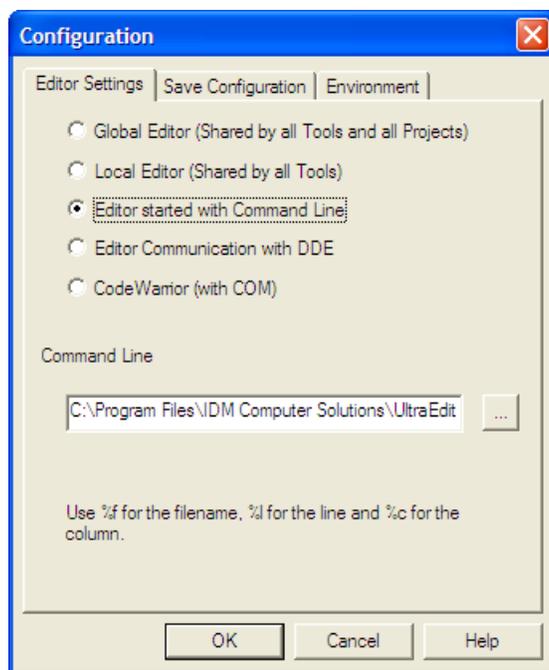


Figure 3-6. Command Line Editor Configuration

3.3.3.1 Example of Configuring a Command Line Editor

The following case 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.

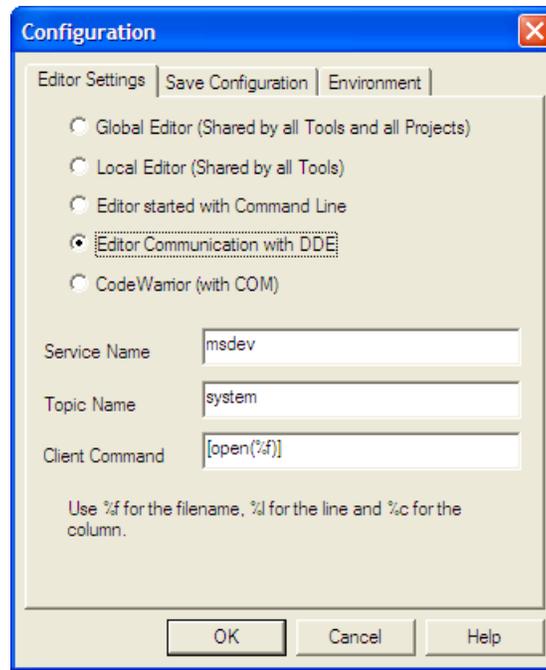


Figure 3-7. DDE Editor Configuration

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.

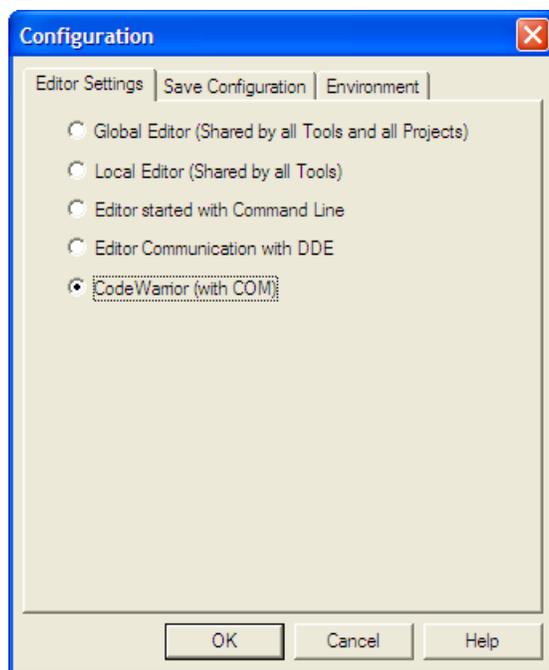


Figure 3-8. COM Editor Configuration

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\WINEDIT\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.

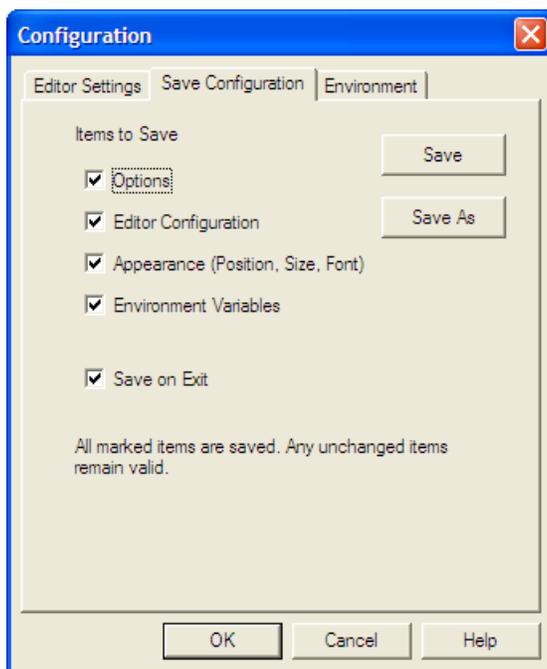


Figure 3-9. Save Configuration Dialog Box

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.

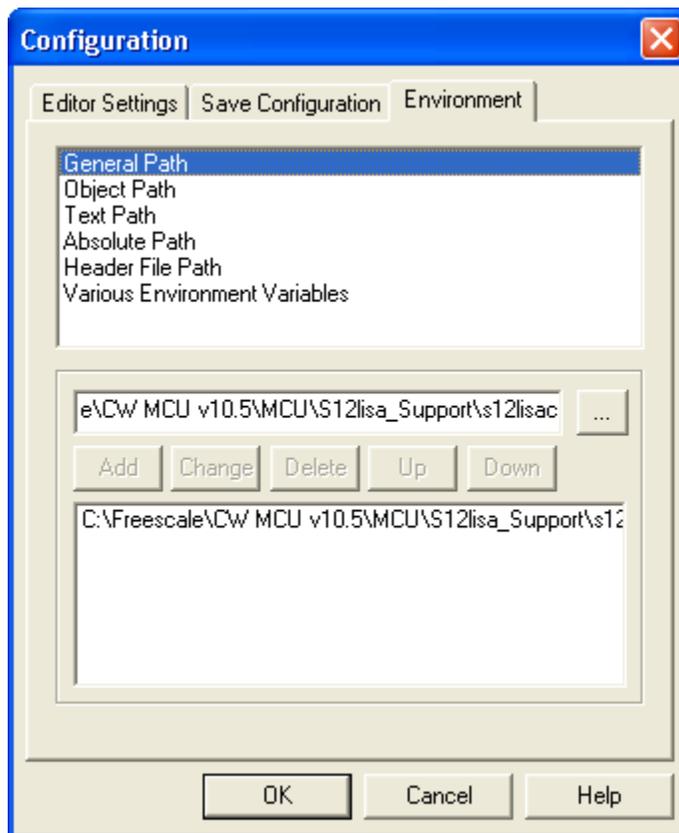


Figure 3-10. Environment Configuration Dialog Box

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 **S12Z Assembler Option Settings** dialog box.

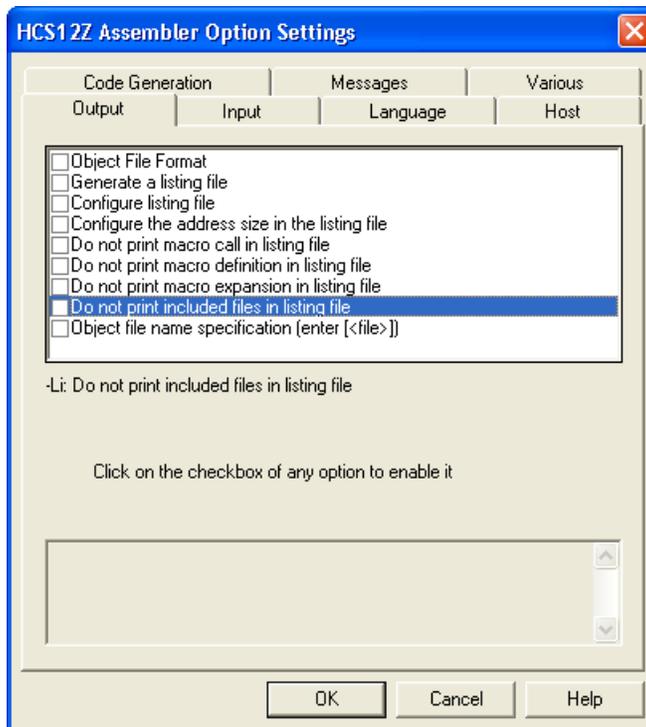


Figure 3-11. Option Settings dialog box

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** dialog box to map messages to a different message class.

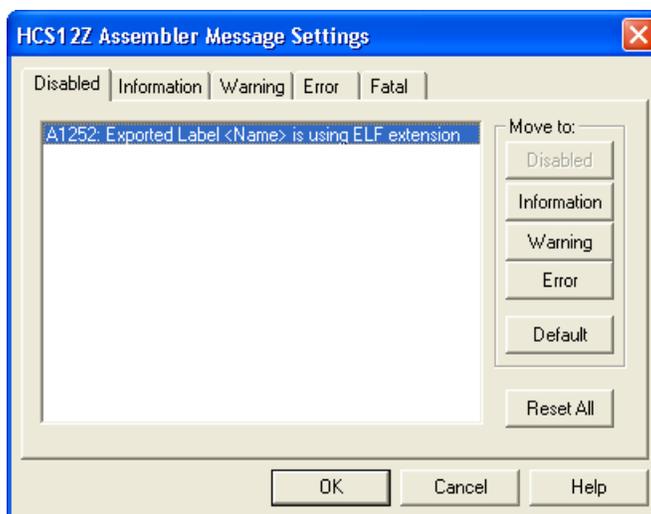


Figure 3-12. Message Settings Dialog Box

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 3-13. 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\fiborr.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 `DEFAULTDIRSYSTEM` 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

CodeWarrior Development Studio for Microcontrollers V10.x S12Z Assembler Reference Manual, Rev.
10.6, 02/2014

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 listed 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 (for more information, refer to the topic [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.

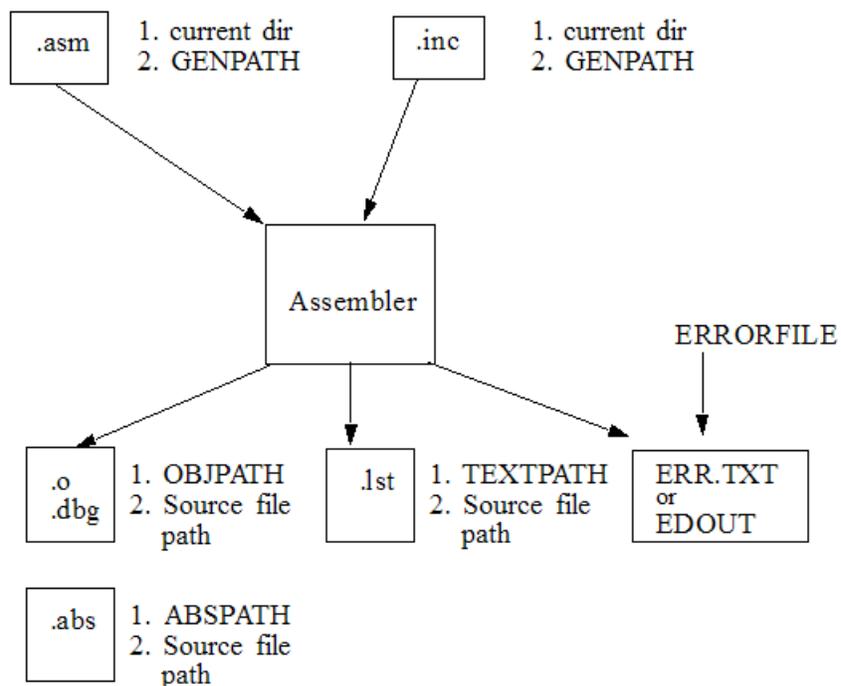


Figure 5-1. Files used with the Assembler



Chapter 6

Assembler Options

This chapter describes the assembler options available for S12Z derivatives. The following topics are covered here:

- [Types of Assembler Options](#)
- [Assembler Option Details](#)
- [List of Assembler Options](#)
- [Detailed Listing of all Assembler Options](#)

6.1 Types of Assembler Options

The Assembler offers a number of assembler options that you can use to control the Assembler's operation. Options are composed of a hyphen (-) followed by one or more letters or digits. Anything not starting with a hyphen is supposed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the [ASMOPTIONS: Default assembler options](#) (refer to the table listed below) environment variable. Typically, each Assembler option is specified only once per assembling session.

Command-line options are not case-sensitive. For example, `-Li` is the same as `-li`. It is possible to combine options in the same group, i.e., one might write `-LcLi` instead of `-Lc -Li`. However such a usage is not recommended as it makes the command line less readable and it does also create the danger of name conflicts. For example `-Li -Lc` is not the same as `-Lic` because this is recognized as a separate, independent option on its own.

NOTE

It is not possible to combine options in different groups, e.g., `-LC -w1` cannot be abbreviated by the terms `-LC1` or `-LCW1`.

Table 6-1. ASMOPTIONS environment variable

ASMOPTIONS	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.
------------	--

Assembler options (refer to the table listed below) are grouped by:

- Output,
- Input,
- Language,
- Host,
- Code Generation,
- Messages, and
- Various.

Table 6-2. Assembler option categories

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 options.

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

Each option has also a scope. The following table lists the scopes for assembler options.

Table 6-3. Scopes for assembler options

Scope	Description
Application	This option has to be set for all files (assembly units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.

Table continues on the next page...

Table 6-3. Scopes for assembler options (continued)

Scope	Description
Assembly Unit	This option can be set for each assembling unit for an application differently. Mixing objects in an application is possible.
None	The scope option is not related to a specific code part. A typical example are options for the message management.

The options available are arranged into different groups, and a tab selection is available for each of these groups. The content of the list box depends upon the tab that is selected.

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 6-4. 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 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 6-5. Special modifiers for assembler options

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

6.2.1.1 Examples using special modifiers

The assumed path and filename (filename base for the modifiers) used for the examples Listing 5.2 through Listing 5.13 is 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 the listing, *Example filename and path used for the following examples* listed above 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

List of Assembler Options

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 Options

The following table lists the command line options you can use with the S12Z Assembler.

NOTE

Not all tools options have been defined for this release. All descriptions will be available in an upcoming release.

Table 6-6. 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
-D: Define Label
-DefLabel: Improves support for data allocation directives
-Env: Set environment variable
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format
-H: Short Help
-I: Include file path
-L: Generate a listing file
-Lasmc: Configure listing file

Table continues on the next page...

Table 6-6. Assembler Options (continued)

Assembler option
-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
-MacroNest: Configure maximum macro nesting
-MCUasm: Switch compatibility with MCUasm ON
-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
-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

Table continues on the next page...

Table 6-6. Assembler Options (continued)

Assembler option
-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

6.4 Detailed Listing of All Assembler Options

The remainder of the chapter is a detailed listing of all assembler options arranged in alphabetical order.

6.4.1 -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`

6.4.2 -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.

6.4.3 -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

6.4.4 -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](#)

6.4.5 -CMacBrackets: Square brackets for macro arguments grouping

Group

Language

Scope

Application

Syntax

-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](#)

6.4.6 -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.

6.4.7 -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 "
      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
    
```

6.4.8 -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

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](#)

6.4.10 -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
```

6.4.11 -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  
  
...
```

6.4.12 -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
```

6.4.13 -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
        LD D0, \1

        ST D0, \2

        ENDM
    
```

Listing: Assembly output listing

```

Abs. Rel.   Loc   Obj. code  Source line
-----
    
```

Detailed Listing of All Assembler Options

```

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 LD D0, \1
8 3i ST D0, \2
9 4i ENDM
10 6 CodeSec: SECTION
11 7 Start:
12 8 cpChar char1, char2
13 2m 000000 xx xxxx + LD D0, char1
14 3m 000003 xx xxxx + ST D0, char2
15 9 000006 01 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](#)

6.4.14 -Lasmc: Configure listing file

Group

Output

Scope

Assembly unit

Syntax

```
-Iasmc={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
```

Detailed Listing of All Assembler Options

```

-----
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](#)

6.4.15 -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`

Detailed Listing of All Assembler Options

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

6.4.16 -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

```

        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 the macro.inc file

```

cpChar:  MACRO
        LD D0, \1

        ST D0, \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

Detailed Listing of All Assembler Options

```

7    2i                                LD D0, \1
8    3i                                ST D0, \2
9    4i                                ENDM
10   6                                CodeSec: SECTION
11   7                                Start:
13   2m 000000 xx xxxx +             LD D0, char1
14   3m 000003 xx xxxx +             ST D0, char2
15   9    000006 01                    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](#)

6.4.17 -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
        LD D0, \1

        ST D0, \2

    ENDM
    
```

Listing: Example assembler output listing

```

Abs. Rel.  Loc   Obj. code  Source line
-----
    
```

Detailed Listing of All Assembler Options

```

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 xx xxxx + LD D0, char1
14 3m 000003 xx xxxx + ST D0, char2
15 9 000006 01 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](#)

6.4.18 -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

```
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 included file

```
cpChar:  MACRO
        LD D0,  \1
```

```

        ST D0, \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			LD D0, \1
8	3i			ST D0, \2
9	4i			ENDM
10	6			CodeSec: SECTION
11	7			Start:
12	8			cpChar char1, char2
15	9	000006 01		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](#)

6.4.19 -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

CodeWarrior Development Studio for Microcontrollers V10.x S12Z Assembler Reference Manual, Rev.
10.6, 02/2014

Detailed Listing of All Assembler Options

```

cpChar:  MACRO
         LD D0,  \1

         ST D0,  \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	xx xxxx +	LD D0, char1
14	3m	000003	xx xxxx +	ST D0, char2
15	9	000006	01	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](#)

6.4.20 -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](#)

6.4.21 -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](#)

6.4.22 -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](#)

6.4.23 -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

6.4.24 -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 S12Z_SHORT;Insert your data
definition here
tmp:          DS.B    1..
FiboLoop:    ST D0,    tmp ; store last
RTS
Test.lst:
Freescale Assembler
Ind.  Name          tiny      short
----  ----          ----      -
1    tmp              1          1
```

6.4.25 -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.

6.4.26 Message A1004 (available in the Online Help)

6.4.27 -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

```
ASMOPTIONS=-MCUasm
```

6.4.28 -N: Display notify box

Group

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.

6.4.29 -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
```

6.4.30 -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
```

6.4.31 -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

6.4.32 -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
fibo.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

6.4.33 -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

6.4.34 -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

6.4.35 -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
```

6.4.36 -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
```

6.4.37 -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
```

6.4.38 -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
```

6.4.39 -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](#)

6.4.40 -Wmsg8x3: Cut filenames in Microsoft format to 8.3

Group

Messages

Scope

Assembly Unit

Syntax

```
-Wmsg8x3
```

Default

None

Description

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

6.4.41 -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.

6.4.42 -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.

6.4.43 -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.

6.4.44 -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.

6.4.45 -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.

6.4.46 -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.
```

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](#)

6.4.47 -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

Detailed Listing of All Assembler Options

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](#)

6.4.48 -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

6.4.49 -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

6.4.50 -WmsgFonf: Message format for no file information

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFonf<string>
```

Arguments

<string>: format string (see below)

Default

```
-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

6.4.51 -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
%p	Path	<code>x:\Freescale\</code>
%f	Path and name	<code>x:\Freescale\sourcefile</code>
%n	Filename	<code>sourcefile</code>
%e	Extension	<code>.asmx</code>
%N	File (8 chars)	<code>sourcefi</code>
%E	Extension (3 chars)	<code>.asm</code>
%K	Uppercase kind	<code>ERROR</code>
%k	Lowercase kind	<code>error</code>
%d	Number	<code>L10324</code>
%m	Message	<code>text</code>
%%	Percent	<code>%</code>
\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

6.4.52 -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](#)

6.4.53 -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](#)

6.4.54 -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.
- **a:** 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
```

6.4.55 -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](#)

6.4.56 -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

6.4.57 -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

6.4.58 -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](#)

6.4.59 -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](#)

6.4.60 -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

```
-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](#)

6.4.61 -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 (for more information, refer to the topic [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:

      LD D0,cst1  ; Loads value in cst1
      ADD D0,cst2 ; Adds value in cst2
      ST D0,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:

        LD D0,cst1  ; Load value into cst1
        ADD D0, cst2 ; Add value in cst2
        ST D0, 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

```
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 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

```
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;
```

relocatable vs. absolute sections

```

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](#).

This chapter covers the following topics:

- [Comment line](#)
- [Source line](#)
- [Symbols](#)
- [Constants](#)
- [Operators](#)
- [Expression](#)
- [Translation limits](#)

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.

This section covers the following topics:

- [Label field](#)
- [Operation field](#)
- [Operand Field: Addressing Modes](#)

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 M68HC12 processor are defined in the CPU Reference Manual (CPU12RM/AD) (http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU12RM.pdf). The instructions for the HCS12X processor are defined in the CPU Reference Manual (S12XCPUV1) (http://www.freescale.com/files/microcontrollers/doc/ref_manual/S12XCPUV1.pdf).

8.2.2.1.1 HCS12Z Instruction Set

The following table presents an overview of the instructions available for HCS12Z. The operands are described in the chapter [Symbols and Notation](#).

Table 8-1. HCS12Z Instruction Set

Instruction	Addressing modes	Descriptions
ABS Di	Inherent	Absolute Value
ADC Di,#oprimmsz	IMM 1/2/4	Add with Carry

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
ADC Di,oprmemreg	OPR 1/2/3	
ADD Di,#oprimsz	IMM 1/2/4	Add without Carry
ADD Di,oprmemreg	OPR 1/2/3	
ADD Di,#oprimsz	IMM 1/2/4	Bitwise AND
ADD Di, oprmemreg	OPR 1/2/3	
ANDCC #opr8i	IMM1	Bitwise AND CCL with Immediate
ASL Dd,Ds,Dn	REG-REG	Arithmetic Shift Left
ASL Dd,Ds,#opr5i	REG-IMM (1-bit, or 5-bit)	
ASL Dd,Ds,oprmemreg	REG-OPR/1/2/3	
ASL.bwpl	OPR/1/2/3-IMM (1-bit, or 5-bit)	
Dd,oprmemreg,#opr5i	OPR/1/2/3-OPR/1/2/3	
ASL.bwpl	REG-IMM (2-operand)	
Dd,oprmemreg,oprmemreg	OPR/1/2/3-IMM (2-operand)	
ASL Di,#opr1i		
ASL.bwpl oprmemreg,#opr1i		
ASR Dd,Ds,Dn	REG-REG	Arithmetic Shift Right
ASR Dd,Ds,#opr5i	REG-IMM	
ASR.bwpl	OPR/1/2/3-IMM	
Dd,oprmemreg,#opr5i ASR.bwpl	OPR/1/2/3-OPR/1/2/3	
Dd,oprmemreg,oprmemreg	OPR/1/2/3-IMM	
ASR.bwpl oprmemreg,#opr1i		
BCC oprdest	REL	Branch if Carry Clear
BCLR Di,#opr5iBCLR	REG-IMM	Test and Clear Bit
Di,DnBCLR.bwl	REG-REG	
oprmemreg,#opr5iBCLR.bwl	OPR/1/2/3-IMM	
oprmemreg,Dn	OPR/1/2/3-REG	
BCS oprdest	REL	Branch if Carry Set
BEQ oprdest	REL	Branch if Equal
BFEXT Dd,Ds,DpBFEXT	REG-REG-REG	Bit Field Extract
Dd,Ds,#width:offset	REG-REG-IMM	
BFEXT.bwplDd,oprmemreg,Dp	REG-OPR/1/2/3-REG	
BFEXT.bwploprmemreg,Ds,Dp	OPR/1/2/3-REG-REG	
BFEXT.bwplDd,oprmemreg,#width: offset	REG-OPR/1/2/3-IMM	
BFEXT.bwploprmemreg,Ds,#width: offset	OPR/1/2/3-REG-IMM	
BFINS Dd,Ds,DpBFINS	REG-REG-REG	Bit Field Insert
Dd,Ds,#width:offset	REG-REG-IMM	
BFINS.bwplDd,oprmemreg,Dp	REG-OPR/1/2/3-REG	
BFINS.bwploprmemreg,Ds,Dp		
BFINS.bwplDd,oprmemreg,#width:		

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
offset BFINS.bwploprmemreg,Ds,#width: offset	OPR/1/2/3-REG-REG REG-OPR/1/2/3-IMM OPR/1/2/3-REG-IMM	
BGE oprdest	REL	Branch if greater than or equal to
BGND	INH	Enter background debug mode
BGT oprdest	REL	Branch if greater than
BHI oprdest	REL	Branch if higher
BHS oprdest	REL	Branch if higher or same
BIT Di,#oprimszBIT Di,oprmemreg	IMM1/2/4 OPR/1/2/3	Bit Test
BLE oprdest	REL	Branch if less than or equal to
BLO oprdest	REL	Branch if lower
BLSooprdest	REL	Branch if lower or same
BLT oprdest	REL	Branch if less than
BMI oprdest	REL	Branch if Minus
BNE oprdest	REL	Branch if Not Equal
BPL oprdest	REL	Branch if Plus
BRA oprdest	REL	Branch always
BRCLR Di,#opr5i,oprdest BRCLR Di,Dn,oprdest BRCLR.bwloprmemreg,#opr5i, oprdest BRCLR.bwloprmemreg,Dn,oprdest	REG-IMM-REL REG-REG-REL OPR/1/2/3-IMM-REL OPR/1/2/3-REG-REL	Test Bit and Branch if Clear
BRSET Di,#opr5i,oprdest BRSET Di,Dn,oprdest BRSET.bwloprmemreg,#opr5i,oprdest BRSET.bwloprmemreg,Dn,oprdest	REG-IMM-REL REG-REG-REL OPR/1/2/3-IMM-REL OPR/1/2/3-REG-REL	Test Bit and Branch if set
BSET Di,#opr5i BSET Di,Dn BSET.bwl oprmemreg,#opr5i BSET.bwl oprmemreg,Dn	REG-IMM REG-REG OPR/1/2/3-IMM OPR/1/2/3-REG	Test and Set Bit
BSR oprdest	REL	Branch to subroutine
BTGL Di,#opr5i BTGL Di,Dn BTGL.bwploprmemreg,#opr5i BTGL.bwploprmemreg,Dn	REG-IMM REG-REG OPR/1/2/3-IMM OPR/1/2/3-REG	Test and Toggle Bit
BVC oprdest	REL	Branch if overflow clear
BVS oprdest	REL	Branch if overflow set

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
CLB cpureg-cpureg	REG-REG	Count leading bits
CLC	IMM1	Clear Carry (Translates to ANDCC # \$FE)
CLI	IMM1	Clear Interrupt Mask (Translates to ANDCC # \$EF)
CLR.bwpl oprmemreg CLR Di CLR X CLR Y	OPR/1/2/3 INH INH INH	Clear Memory, Register, or Index Register
CLV	IMM1	Clear Overflow (Translates to ANDCC # \$FD)
CMP Di,#oprimszCMP Di,oprmemreg CMP xy,#opr24i CMP xy,oprmemreg CMP S,#opr24i CMP S,oprmemreg CMP X,Y	IMM1/2/4 OPR/1/2/3 IMM3 OPR/1/2/3 IMM3 OPR/1/2/3 INH	Compare
COM.bwl oprmemreg	OPR/1/2/3	Complement memory
DBcc Di,oprdest DBcc X,oprdest DBcc Y,oprdest DBcc.bwploprmemreg,oprdest	REG-REL REG-REL REG-REL OPR/1/2/3-REL	Decrement and Branch
DEC Di DEC.bwl oprmemreg	INH OPR/1/2/3	Decrement
DIVS Dd,Dj,Dk DIVS.B Dd,Dj,#opr8i DIVS.W Dd,Dj,#opr16i DIVS.L Dd,Dj,#opr32i DIVS.bwl Dd,Dj,oprmemreg DIVS.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Signed Divide
DIVU Dd,Dj,Dk DIVU.B Dd,Dj,#opr8i DIVU.W Dd,Dj,#opr16i DIVU.L Dd,Dj,#opr32i DIVU.bwl Dd,Dj,oprmemreg DIVU.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Unsigned divide
EOR Di,#oprimsz	IMM1/2/4	Exclusive OR

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
EOR Di, oprmemreg	OPR/1/2/3	
EXG cpureg, cpureg	INH	Exchange register contents
INC Di	INH	Increment
INC.bwl oprmemreg	OPR/1/2/3	
JMP opr24a	EXT3	Jump
JMP oprmemreg	OPR/1/2/3	
JSR opr24a	EXT3	Jump to subroutine
JSR oprmemreg	OPR/1/2/3	
LD Di, #oprimsz	IMM1/2/4 (same size as Di)	Load (Di, X, Y, or SP)
LD Di, opr24a	EXT3 (24-bit address)	
LD Di, oprmemreg	OPR/1/2/3 IMM2 (efficient 18-bit)	
LD xy, #opr18i	IMM3 (same size as X or Y)	
LD xy, #opr24i	EXT3 (24-bit address)	
LD xy, opr24a	OPR/1/2/3 IMM3 (same size as SP)	
LD xy, oprmemreg	OPR/1/2/3	
LD S, #opr24i		
LD S, oprmemreg		
LEA D67, oprmemreg	OPR/1/2/3	Load Effective Address
LEA S, oprmemreg	OPR/1/2/3	
LEA xy, oprmemreg	OPR/1/2/3	
LEA S, (#opr8i, S)	IMM1 (8-bit signed offset)	
LEA xy, (#opr8i, xy)	IMM1 (8-bit signed offset)	
LSL Dd, Ds, Dn	REG-REG	Logical Shift Left
LSL Dd, Ds, #opr5i	REG-IMM	
LSL.bwpl Dd, oprmemreg, #opr5i	OPR/1/2/3-IMM	
LSL.bwpl Dd, oprmemreg, oprmemreg	OPR/1/2/3-OPR/1/2/3	
LSL.bwpl oprmemreg, #opr1i	OPR/1/2/3-IMM	
LSR Dd, Ds, Dn	REG-REG	Logical Shift Right
LSR Dd, Ds, #opr5i	REG-IMM	
LSR.bwpl Dd, oprmemreg, #opr5i	OPR/1/2/3-IMM	
LSR.bwpl Dd, oprmemreg, oprmemreg	OPR/1/2/3-OPR/1/2/3	
LSR.bwpl oprmemreg, #opr1i	OPR/1/2/3-IMM	
MACS Dd, Dj, Dk	REG-REG	Signed Multiply and Accumulate
MACS.B Dd, Dj, #opr8i	REG-IMM1	
MACS.W Dd, Dj, #opr16i	REG-IMM2	
MACS.L Dd, Dj, #opr32i	REG-IMM4	
	REG-OPR/1/2/3	

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
MACS.bwl Dd,Dj,oprmemreg MACS.bwplbwplDd,oprmemreg,oprmemreg	OPR/1/2/3-OPR/1/2/3	
MACU Dd,Dj,Dk MACU.B Dd,Dj,#opr8i MACU.W Dd,Dj,#opr16i MACU.L Dd,Dj,#opr32i MACU.bwl Dd,Dj,oprmemreg MACU.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Unsigned multiply and accumulate
MAXS Di,oprmemreg	OPR/1/2/3	Maximum of two signed values to Di
MAXU Di,oprmemreg	OPR/1/2/3	Maximum of two unsigned values to Di
MINS Di,oprmemreg	OPR/1/2/3	Minimum of two signed values to Di
MINU Di,oprmemreg	OPR/1/2/3	Minimum of two unsigned values to Di
MODS Dd,Dj,Dk MODS.B Dd,Dj,#opr8i MODS.W Dd,Dj,#opr16i MODS.L Dd,Dj,#opr32i MODS.bwl Dd,Dj,oprmemreg MODS.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Signed Modulo
MODU Dd,Dj,Dk MODU.B Dd,Dj,#opr8i MODU.W Dd,Dj,#opr16i MODU.L Dd,Dj,#opr32i MODU.bwl Dd,Dj,oprmemreg MODU.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Unsigned Modulo
MOV.B #opr8i,oprmemreg MOV.W #opr16i,oprmemreg MOV.P #opr24i,oprmemreg MOV.L #opr32i,oprmemreg MOV.bwpl oprmemreg,oprmemreg	IMM1-OPR/1/2/3 IMM2-OPR/1/2/3 IMM3-OPR/1/2/3 IMM4-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Move Data (8, 16, 24, or 32-bits; IMM-OPR or OPR-OPR)
MULS Dd,Dj,Dk MULS.B Dd,Dj,#opr8i MULS.W Dd,Dj,#opr16i MULS.L Dd,Dj,#opr32i MULS.bwl Dd,Dj,oprmemreg MULS.bwplbwplDd	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Signed Multiply

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
, oprmemreg, oprmemreg		
MULU Dd,Dj,Dk MULU.B Dd,Dj,#opr8i MULU.W Dd,Dj,#opr16i MULU.L Dd,Dj,#opr32i MULU.bwl Dd,Dj,oprmemreg MULU.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Unsigned multiply
NEG.bwl oprmemreg	OPR/1/2/3	Two's complement negate
NOP	INH	Null operation
OR Di,#oprimsz OR Di,oprmemreg	IMM1/2/4 OPR/1/2/3	Bitwise OR
ORCC #opr8i	IMM1	Bitwise OR with CCL Immediate
PSH oprregs1 PSH oprregs2 PSH ALL PSH ALL16b	INH INH INH INH	Push registers onto stack
PUL oprregs1 PUL oprregs2 PUL ALL PUL ALL16b	INH INH INH INH	Pull registers from stack
QMULS Dd,Dj,Dk QMULS.B Dd,Dj,#opr8i QMULS.W Dd,Dj,#opr16i QMULS.L Dd,Dj,#opr32i QMULS.bwl Dd,Dj,oprmemreg QMULS.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Signed Fractional multiply
QMULU Dd,Dj,Dk QMULU.B Dd,Dj,#opr8i QMULU.W Dd,Dj,#opr16i QMULU.L Dd,Dj,#opr32i QMULU.bwl Dd,Dj,oprmemreg QMULU.bwplbwplDd,oprmemreg,oprmemreg	REG-REG REG-IMM1 REG-IMM2 REG-IMM4 REG-OPR/1/2/3 OPR/1/2/3-OPR/1/2/3	Unsigned Fractional multiply
ROL.bwpl oprmemreg	OPR/1/2/3	Rotate left through carry
ROR.bwpl oprmemreg	OPR/1/2/3	Rotate right through carry
RTI	INH	Return from Interrupt

Table continues on the next page...

Table 8-1. HCS12Z Instruction Set (continued)

Instruction	Addressing modes	Descriptions
RTS	INH	Return from subroutine
SAT Di	INH	Saturate
SBC Di, #oprimsz	IMM1/2/4	Subtract with Borrow
SBC Di, oprmemreg	OPR/1/2/3	
SEV	IMM1	Set Carry Flag (Translates to ORCC # \$01)
SEI	IMM1	Set Interrupt Mask (Translates to ORCC # \$10)
SEV	IMM1	Set Overflow flag (Translates to ORCC # \$02)
SEX cpureg, cpureg	INH	Sign extend (smaller CPU register to a larger CPU register)
ST Di, opr24a	EXT (24-bit address)	Store (Di, X, Y or SP)
ST Di, oprmemreg	OPR/1/2/3	
ST xy, opr24a	EXT (24-bit address)	
ST xy, oprmemreg	OPR/1/2/3	
ST S, oprmemreg	OPR/1/2/3	
STOP	INH	Stop processing
SUB Di, #oprimsz	IMM1/2/4	Subtract without borrow
SUB Di, oprmemreg	OPR/1/2/3	
SUB D6, X, Y	INH	
SUB D6, Y, X	INH	
SWI	INH	Software interrupt
SYS	INH	System call software interrupt
TBcc Di, oprdest	REG-REL	Test and branch
TBcc X, oprdest	REG-REL	
TBcc Y, oprdest	REG-REL	
TBcc.bwpoprmemreg, oprdest	OPR/1/2/3-REL	
TFR cpureg, cpureg	INH	Transfer register contents
TRAP #num	INH	Unimplemented Page2 Opcode Trap
WAI	INH	Wait for interrupt
ZEX cpureg, cpureg	INH	Zero-extend (smaller CPU register to a larger CPU register)

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

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 8-2. HCS12Z Addressing Mode Notation

Addressing Mode	Notation	Example
Inherent	No operand	RTS
Register	INST Di	ADD D0,D1
Immediate	INST #oprimsz	ADD D2,#45
Relative	INST oprdest	JSR label
Extended (14 bit address)	INST opru14	ADD D4,100
Extended (18 bit address)	INST opru18	ADD D5,65536
Extended (24 bit address)	INST opru24	ADD D7,4194304
Indexed - IDX	INST (opru4,XYs)	ADD D1,(10,X)
Indexed - IDX1	INST (oprs9,XYSP)	ADD D4,(254, X)
Indexed - IDX3	INST (opr24,XYSP)	ADD D6,(8388604, X)
Indexed Indirect	INST [Di,XYs]	ADD D0,[D0,X]
Indexed Indirect - s9	INST [oprs9,XYSP]	ADD D3,[254, X]
Indexed Indirect - 24b	INST [opr24,XYSP]	ADD D7,[8388000, Y]
Extended 3	INST opr24	JSR 8388000
Indexed Indirect Extended 3	INST [opr24]	LD D6,[4552]

NOTE

For more information on the operands specified in **Notation** column, refer to the chapter [Symbols and Notation](#).

This section includes the following topics:

source line

- [Inherent Addressing Mode](#)
- [Register Addressing Mode](#)
- [Immediate Addressing Mode](#)
- [Short Immediate Addressing Mode \(IMMe4\)](#)
- [Relative Addressing Modes \(REL, REL1\)](#)
- [Extended Addressing Modes \(EXT1, EXT2, EXT3, EXT24\)](#)
- [Indexed Addressing Modes](#)
- [Indexed Indirect Addressing Modes](#)

8.2.3.1 Inherent Addressing Mode

Instructions using this addressing mode have no operands or all operands are stored in internal CPU registers. The CPU does not need to perform any memory access to complete the instruction.

Listing: Inherent Addressing Mode Instructions

```
NOP ; Instruction with no operand
ABS D0 ; Returns the absolute of value present in D0 register.
```

8.2.3.2 Register Addressing Mode

The operand is one of the eight CPU data registers (Di) and so no memory access is needed.

The register number 0-7 is encoded in the opcode or an instruction postbyte.

Listing: Register Addressing Mode

```
main: ADD D0,D1
      AND D6,D7
```

8.2.3.3 Immediate Addressing Mode

The opcode contains the value to use with the instruction rather than the address of this value. The '#' character is used to indicate an immediate addressing mode operand. The immediate value can be of 1,2,3, or 4 bytes.

The effective address of the instruction is specified using the # character, as listed in the following listing:

Listing: Immediate Addressing Mode

```
main: LD D2,#100
      CMP D2,#100

      BNE main
```

8.2.3.4 Short Immediate Addressing Mode (IMMe4)

A 4-bit immediate operand is encoded in the instruction to provide a very efficient way to initialize registers or variables with the common values -1, 1, 2, 3,...13, 14, or 15 (automatically sign-extended to the required size).

Listing: Short Immediate Addressing Mode

```
main: LD D0,#10
      CMP D0,#10

      BNE main
```

8.2.3.5 Relative Addressing Modes (REL, REL1)

A 7-bit (or 15-bit) two's complement relative offset is included in the instruction opcode. The relative offset is computed by adding the signed offset to the address of the first byte of object code for the current instruction.

Listing: Relative Addressing Modes

```
main: LD D0,#10
      CMP D0,#10

      BEQ foo
```

...

foo:

REL addressing mode comprises of 7-bit signed relative offset and REL1 mode has 15-bit signed relative offset.

8.2.3.6 Extended Addressing Modes (EXT1, EXT2, EXT3, EXT24)

A 14-bit, 18-bit, or 24-bit address of the operand is provided in the instruction. In the case of 14-bit EXT1 and 18-bit EXT2 addressing modes, the supplied address is zero-extended to 24-bits to form the address of the operand.

Listing: Extended Addressing Modes

```
Main: LD X,8388000
      BHS label1
```

8.2.3.7 Indexed Addressing Modes

This addressing mode adds a 4-bit(9-bit or 24-bit) unsigned offset to the base index register to form the memory address that is referenced in the instruction. The valid range for a 4-bit unsigned offset is [0..15]. The base index register may be X, Y, SP, PC.

These indexed addressing modes use an index register as a base address and add a constant or register offset to form the effective address of the operand. The index register is usually X, Y, SP, or PC, but in a few modes a CPU data register Di can be used as the index base address. IDX implies zero extension bytes which means everything the instruction needs is included in the instruction or internal CPU registers itself.

IDX1, IDX2, and IDX3 imply 1, 2, or 3 additional extension bytes are needed, respectively.

The topics covers here are as follows:

- [4-Bit Short Constant Offset from X, Y, or SP \(IDX\)](#)
- [9-Bit Constant Offset from X, Y, SP or PC \(IDX1\)](#)
- [24-Bit Constant Offset from X, Y, SP or PC \(IDX3\)](#)
- [Register Offset Indexed from X, Y, or SP \(REG,IDX\)](#)
- [Automatic Pre/Post Increment/Decrement from X, Y, or SP \(++IDX\)](#)
- [18-Bit Constant Offset from Di \(IDX2,REG\)](#)
- [24-Bit Constant Offset from Di \(IDX3,REG\)](#)

8.2.3.7.1 4-Bit Short Constant Offset from X, Y, or SP (IDX)

A 4-bit unsigned constant (0-15) is added to X, Y, or SP to form the effective address of the operand. This addressing mode is very compact and efficient and handles the most common indexed addressing offsets.

Larger offsets are supported with other indexed addressing mode variations which use additional extension bytes to specify the larger offsets.

Listing: Example - 4-Bit Short Constant Offset

```

ORG $1000
CST_TBL: DC.B $5, $10, $18, $20, $28, $30

ORG $800

DATA_TBL: DS.B 10

main:

    LD X, #CST_TBL

    LD D0, (3, X)

    LD Y, #DATA_TBL

    ST D0, (8, Y)

```

Register D0 is loaded with byte value stored in memory location \$1003 (\$1000 + 3) and the contents of D0 are stored to memory \$808 (\$800 + 8).

8.2.3.7.2 9-Bit Constant Offset from X, Y, SP or PC (IDX1)

A 9-bit signed constant (-256 to +255) is added to X, Y, SP or PC to form the effective address of the operand.

Listing: Example - 9-Bit Constant Offset

```

ORG $1000
CST_TBL: DC.B $5, $10, $18, $20, $28, $30, $38, $40, $48

DC.B $50, $58, $60, $68, $70, $78, $80, $88, $90

DC.B $98, $A0, $A8, $B0, $B8, $C0, $C8, $D0, $D8

ORG $800

DATA_TBL: DS.B 40

main:

    LD X, #CST_TBL

    LD D2, (20, X)

    LD Y, #DATA_TBL

    ST D2, (18, Y)

```

8.2.3.7.3 24-Bit Constant Offset from X, Y, SP or PC (IDX3)

Source line

A 24-bit constant is added to X, Y, SP or PC to form the effective address of the operand.

Listing: Example - 24-Bit Constant Offset

```
main:
LD X,#$600

LD D6,($300,X)

LD Y,$1000

ST D6,($140, Y)
```

8.2.3.7.4 Register Offset Indexed from X, Y, or SP (REG,IDX)

A CPU data registers D_i is added to X, Y, or SP to form the effective address of the operand. This indexed addressing sub-mode allows a program-controlled offset which can change during execution of the program. For registers D0, D1, D6, and D7 the register is treated as an unsigned value. For D2~D5 the register is treated as a signed value.

Listing: Example - Register Offset Indexed

```
main:
LD D0,$300

LD D1,$400

LD X,$600

LD D6,(D0,X)

LD Y,$1000

ST D6,(D1,Y)
```

8.2.3.7.5 Automatic Pre/Post Increment/Decrement from X, Y, or SP (+IDX)

X, Y, or SP is used to access an operand either before or after it is incremented or decremented. The increment/decrement value is determined by the size of the operand that is being accessed. When SP is used as the index register, only pre-decrement (as in a PUSH) and post-increment (as in a PULL) variations are allowed. When X or Y is used as the index register, all four variations (pre-decrement, pre-increment, post-decrement, and post increment) are supported. In cases where an instruction has more than one operand that uses indexed addressing, any auto-increment or decrement is done during processing of the current operand.

Listing: Example - Automatic Pre/Post Increment/Decrement

```
MOV.W (X+), (D2,X)
```

The CPU would first read the 16-bit memory value pointed to by index register X, then increment X (by 2 because the operand that was read was two bytes), then store the value at the address that is formed by adding D2 to index register X (the new incremented value in X, not the value X had when the instruction started).

8.2.3.7.6 18-Bit Constant Offset from Di (IDX2,REG)

An 18-bit unsigned constant is added to a CPU registers Di to form the effective address of the operand. For registers D0, D1, D6, and D7 the register is treated as an unsigned value. For D2~D5 the register is treated as a signed value.

8.2.3.7.7 24-Bit Constant Offset from Di (IDX3,REG)

A 24-bit constant is added to a CPU registers Di to form the effective address of the operand. For registers D0, D1, D6, and D7 the register is treated as an unsigned value. For D2~D5 the register is treated as a signed value.

8.2.3.8 Indexed Indirect Addressing Modes

These addressing modes use an indexed addressing mode to form the effective address of a pointer to the operand rather than using the indexed addressing mode to get the effective address of the operand itself.

In all cases, the intermediate pointer that is fetched from the effective address is 24 bits and this 24-bit address is used to fetch the operand. The size of the operand (1, 2, 3, or 4 bytes) that this pointer points to, depends on the instruction.

The topics covered here are as follows:

- [Register Offset Indexed Indirect from X or Y \(\[REG,IDX\]\)](#)
- [9-Bit Constant Offset Indexed Indirect from X, Y, SP or PC \(\[IDX1\]\)](#)
- [24-Bit Constant Offset Indexed Indirect from X, Y, SP or PC \(\[IDX3\]\)](#)
- [Address Indirect Addressing Mode \(\[EXT3\]\)](#)

8.2.3.8.1 Register Offset Indexed Indirect from X or Y ([REG,IDX])

Source line

A CPU data registers D_i is added to X or Y to form the effective address of the pointer to the operand. For registers D0, D1, D6, and D7 the register is treated as an unsigned value. For D2~D5 the register is treated as a signed value.

8.2.3.8.2 9-Bit Constant Offset Indexed Indirect from X, Y, SP or PC ([IDX1])

A 9-bit signed constant (-256 to +255) is added to X, Y, SP or PC to form the effective address of the pointer to the operand.

Listing: Example - Indexed Indirect 9-Bit Constant Offset

```
ORG $1000
CST_TBL1: DC.W $1020, $1050, $2001

ORG $2000

CST_TBL: DC.B $10, $35, $46

ORG $3000

main:

LD X, #CST_TBL1

LD D6, [4,X]
```

The constant offset 4 is added to the value of register X (\$1000) to form the address \$1004. Then an address pointer (\$2001) is read from memory at \$1004. The register D6 is loaded with \$35, the value stored at address \$2001.

8.2.3.8.3 24-Bit Constant Offset Indexed Indirect from X, Y, SP or PC ([IDX3])

A 24-bit constant is added to X, Y, SP or PC to form the effective address of the pointer to the operand.

8.2.3.8.4 Address Indirect Addressing Mode ([EXT3])

This addressing mode uses a 24-bit constant to point to a pointer which is then used to access the operand. This allows a 24-bit pointer to an operand to be located anywhere in the 16-megabyte memory space. The 24-bit constant address that points to the pointer to the operand is supplied as three extension bytes in the object code of the instruction.

Listing: Example - Address Indirect Addressing Mode

```
ORG $1100
Cnst_tbl DC.W $10,$11,$12,$13

ORG $1114

ptr DS 3

ORG $FF0000

main:
    MOV.P    #Cnst_tbl,$1114
    LD      D2,[$1114]
    CMP     D2,#10
```

`ptr` points to address of `Cnst_tbl`. Then the first value of `Cnst_tbl` array is loaded to `D2` and compared with value `10`.

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

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 S12Z processor the reserved identifiers are listed in the following listing:

Listing: Reserved identifiers for an HCS12Z 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
- 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

Operators

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)

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:
<operand> & <operand>
```

```
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

Operators

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

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 S12Z 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 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 S12Z 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).

8.5.14 Operator precedence

The following table lists the operator precedence rules for ANSI - C operators.

Table 8-3. 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
- 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 8-4. 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 8-5. 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 chapter gives you the overview for assembler directives. The topics covered here are as follows:

- [Section-Definition Directives](#)
- [Constant-Definition directives](#)
- [Data-Allocation directives](#)
- [Symbol-Linkage directives](#)
- [Assembly-Control directives](#)
- [Listing-File Control directives](#)
- [Macro Control directives](#)
- [Conditional Assembly 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
ORG - Set Location Counter	Define an absolute section
SECTION - Declare Relocatable Section	Define a relocatable section
OFFSET - Create absolute symbols	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
EQU - Equate symbol value	Assign a name to an expression (cannot be redefined)
SET - Set Symbol Value	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
DC - Define Constant	Define a constant variable
DCB - Define Constant Block	Define a constant block
DS - Define Space	Define storage for a variable
RAD50 - RAD50-encoded string constants	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
ABSENTRY - Application entry point	Specify the application entry point when an absolute file is generated
XDEF - External Symbol Definition	Make a symbol public (visible from outside)
XREF - External Symbol Reference	Import reference to an external symbol.
XREFB - External Reference for Symbols located on the Direct Page	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
ALIGN - Align Location Counter	Define Alignment Constraint
BASE - Set number base	Specify default base for constant definition
END - End assembly	End of assembly unit
ENDFOR - End of FOR block	End of FOR block
EVEN - Force word alignment	Define 2-byte alignment constraint
FAIL - Generate Error message	Generate user defined error or warning messages
FOR - Repeat assembly block	Repeat assembly blocks
INCLUDE - Include text from another file	Include text from another file.
LONGEVEN - Forcing Long-Word alignment	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
CLIST - List conditional assembly	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable Listing	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set Line Length	Define line length in assembly listing file.
MLIST - List macro expansions	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable Listing	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable Paging	Disable paging in the assembly listing file.
PAGE - Insert Page break	Insert page break.
PLEN - Set Page Length	Define page length in the assembler listing file.

Table continues on the next page...

Table 9-6. Listing-file control directives (continued)

Directive	Description
SPC - Insert Blank Lines	Insert an empty line in the assembly listing file.
TABS - Set Tab Length	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide Listing Title	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
ENDM - End macro definition	End of user defined macro.
MACRO - Begin macro definition	Start of user defined macro.
MEXIT - Terminate Macro Expansion	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
ELSE - Conditional assembly	alternate block
ENDIF - End conditional assembly	End of conditional block
IF - Conditional assembly	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly	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.

Table continues on the next page...

Table 9-8. Conditional assembly directives (continued)

Directive	Description
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined
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

```
4      4                                     base  10  ; default base: decimal
```

Detailed descriptions of all assembler directives

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

        LD D2,    #103

    ELSE

        LD D2,    #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	LD D2, #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

        LD D2,    #103

    ELSE

        LD D2,    #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	LD D2, #103
5	5			ELSE
6	6			LD D2, #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

          LD D2,#103

      ELSE

          LD D2,#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

Abs.	Rel.	Loc	Obj. code	Source line
1	1	0000 0001	Try: EQU	1

Detailed descriptions of all assembler directives

```

2      2      0000 0001      IF Try != 0
3      3      000000 A667      LD D2,#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
         LD D6, \1

         ST D6, \2

         ENDM

CodeSec: SECTION

Start:

        cpChar char1, char2

        LD D6, char1

        ST D6, 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
    LD D6, \1
ENDIF
IFC "\2", ""
    FAIL 600
ELSE
    ST D6, \2
```

Detailed descriptions of all assembler directives

ENDIF

ENDM

codSec: SECTION

Start:

cpChar char1

Listing: Error messages resulting from assembling the source code

```
>> in "C:\Freescale\demo\warnfail.asm", line 9, col 0, pos 99
    FAIL 600
    ^
```

WARNING A2332: FAIL found

INFORMATION Macro Expansion 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

                LD D6, \1

    ENDIF

    IFC "\2", ""

                FAIL 600

    ELSE

                ST D6, \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\err.asm", line 3, col 0, pos 41
    FAIL 200
    ^
```

```
ERROR A2329: FAIL found
```

```
INFORMATION Macro Expansion                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

        LD D6, \1

    ENDIF

    IFC "\2", ""

        FAIL 600

    ELSE

        ST D6, \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\test.asm", line 3, col 0, pos 37
    FAIL "A character must be specified as first parameter"
    ^

ERROR A2338: A character must be specified as first parameter

INFORMATION Macro Expansion                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
          LD D6,#100

      ELSE
          LD D6,#0

      ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `LD D6,#0` instruction is assembled. Changing the operand of the `EQU` directive to one causes the `LD D6,#100` 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
LD D6,#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. 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

Table continues on the next page...

Table 9-9. Conditional assembly types (continued)

ifcc	Condition	Meaning
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

        LD D6,#103

    ELSE

        LD D6,#0

    ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `LD D6,#0` instruction is assembled. Changing the directive to `IFEQ` causes the `LD D6,#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      LD D6,#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

Detailed descriptions of all 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

        LD D6,\1

        ST D6,\2

        ENDM

CodeSec: SECTION

Start:

        cpChar char1, char2

        LD D6,char1

        ST D6,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

```

        XDEF entry
storage: EQU $00FF

save: MACRO ; Start macro definition

        LD X,#storage

        LD D6, \1

        ST D6,(0,x) ; Save first argument

        LD D6, \2

        ST D6, (2,x) ; Save second argument

        IFC '\3', '' ; Is there a third argument?
                MEXIT ; No, exit from macro

        ENDC

        LD D6, \3 ; Save third argument

        ST D6,(4,X)

        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			save: MACRO ; Start macro definition
4	4			LD X,#storage
5	5			LD D6, \1
6	6			ST D6,(0,x) ; Save first argument

```

7      7      LD D6, \2
8      8      ST D6, (2,x) ; Save second argument
9      9      IFC '\3','';Is there a third argument?
10     10     MEXIT ; No, exit from macro
11     11     ENDC
12     12     LD D6, \3 ; Save third argument
13     13     ST D6,( 4,X)
14     14     ENDM ; End of macro definition
15     15     datSec: SECTION
16     16     000000      char1:  ds.b 1
17     17     000001      char2:  ds.b 1
18     18     codSec: SECTION
19     19     entry:
20     20     save char1, char2
21     4m 000000 9800 00FF + LD X,#storage
22     5m 000004 B6xx xxxx + LD D6, char1
23     6m 000008 C640      + ST D6,(0,x) ; Save first argument
24     7m 00000A B6xx xxxx + LD D6, char2
25     8m 00000E C642      + ST D6, (2,x) ; Save second argument
26     9m      0000 0001 + IFC '','';Is there a third argument?
28     10m      + MEXIT ; No, exit from macro
29     11m      + ENDC
30     12m      + LD D6, ;Save third argument
31     13m      + ST D6,( 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.

Detailed descriptions of all assembler directives

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

swap: MACRO

    LD D6, \1

    LD X, \2

    ST D6, \2

    ST X, \1

ENDM

codSec: SECTION

entry:

    LD D6, #F0

    LD X, #0F

main:

    ST D6, first

    ST X, second

    swap first, second

    NOP

    BRA    main

datSec: SECTION

first: DS.B 1

second: DS.B 1

```

Listing: Assembler Output

Abs. Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----

```

1  1          XDEF entry
3  3          swap: MACRO
4  4          LD D6, \1
5  5          LD X, \2
6  6          ST D6, \2
7  7          ST X, \1
8  8
9  9          ENDM
10 10         codSec: SECTION
11 11         entry:
12 12 000000 9600 0000          LD D6, #$F0
          000004 F0
13 13 000005 A87F              LD X, #$0F
14 14         main:
15 15 000007 D6xx xxxx          ST D6, first
16 16 00000B C800 xxxx          ST X, second
          00000F xx
17 17         swap first, second
18  4m 000010 B6xx xxxx +       LD D6, first
19  5m 000014 A800 xxxx +       LD X, second
          000018 xx
20  6m 000019 D6xx xxxx +       ST D6, second
21  7m 00001D C800 xxxx +       ST X, first
          000021 xx
22  8m         +
23 18 000022 01                NOP
24 19 000023 20FF E4           BRA main
25 20         datSec: SECTION
26 21 000000                   first: DS.B 1
27 22 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
-----

```

Detailed descriptions of all assembler directives

```

1 1 XDEF entry
3 3 swap:MACRO
4 4 LD D6, \1
5 5 LD X, \2
6 6 ST D6, \2
7 7 ST X \1
8 8
9 9 ENDM
10 10 codSec: SECTION
11 11 entry:
12 12 000000 9600 0000 LD D6, #$F0
    000004 F0
13 13 000005 A87F LD X, #$0F
14 14 main:
15 15 000007 D6xx xxxx ST D6, first
16 16 00000B C800 xxxx ST X, second
    00000F xx
17 17 swap first, second
23 18 000022 01 NOP
24 19 000023 20FF E4 BRA main
25 20 datSec: SECTION
26 21 000000 first: DS.B 1
27 22 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 listing from the assembler source code

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1	000000	01	aaa: NOP
3	3	000001	01	bbb: NOP
4	4	000002	01	NOP
9	9	000005	01	ddd: NOP
10	10	000006	01	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

Abs.	Rel.	Loc	Obj. code	Source line
-----	-----	-----	-----	-----
1	1			OFFSET 0
2	2	000000	ID:	DS.B 1
3	3	000001	COUNT:	DS.W 1
4	4	000003	VALUE:	DS.L 1

```

5 5          0000 0007  SIZE:  EQU  *
6 6
7 7          DataSec: SECTION
8 8  000000          Struct: DS.B  SIZE
9 9
10 10         CodeSec: SECTION
11 11         entry:
12 12  000000 98xx xxxx          LD X, #Struct
13 13  000004 9600 0000          LD D6, #0
    000008 00
14 14  000009 C640          ST D6, (ID, X)
15 15  00000B 9C41          INC.B (COUNT, X)
16 16  00000D 36          INC D6
17 17  00000E C643          ST D6, (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 listing from the source code

Abs.	Rel.	Loc	Obj. code	Source line
----	----	-----	-----	-----
1	1			ORG \$2000
2	2	a002000	01	b1: nop
3	3	a002001	05	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 listing from the source code

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 (*Listing: Example-Program that Prints Hello World*) listed below about how to decode it.

Example

The string data in the following listing assembles to the following data (*Listing: Assembler output 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	01	xx: NOP
3	3			bbb: SECTION 5
4	4			
5	5	000000	01	yy: NOP
6	6	000001	01	NOP
7	7	000002	01	NOP
8	8			aaa: SECTION 4
9	9	000001	01	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 S12Z, there are two additional section qualifiers: `S12Z_SHORT` and `S12Z_TINY`. When a section is declared as `S12Z_SHORT` (or `S12Z_TINY`) all the objects defined there can be accessed using the short (and respectively tiny) addressing modes.

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                                     ENDF
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 S12Z, 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 S12Z, 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 Macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.

- 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
    LD X, #\1

    LD D6, #16

    \@LOOP: CLR.B (0,X)

    INC.B (0,X)

    DEC D6

    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
        LD X, #temporary

        LD D6, #16
_00001LOOP: CLR.B (0,X)

        INC.B(0,X)

        DEC D6

        BNE _00001LOOP

        clear data

        LD X, #data

        LD D6, #16

_00002LOOP: CLR.B (0,X)

        INC.B (0,X)

        DEC D6

        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 name of the generated listing file is `<base name>.lst`.

The following listed are the assembler options available for listing files.

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

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 - `S12Z`.
- The third line contains a copyright notice.

Listing: Example page header output

```
Demo Application
Freescale S12Z-Assembler

(c) COPYRIGHT Freescale 1997-2012
```

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			LD D0, \1
12	3i			ST D0, \2
13	4i			ENDM
14	10			CodeSec: SECTION
15	11			Start:
16	12			cpChar char1, char2
17	2m	000000	xx xxxx	+ LD D0, char1
18	3m	000003	xx xxxx	+ ST D0, char2
19	13	000006	01	NOP
20	14	000007	01	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 code listed 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			-----
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			LD D0, \1
12	3i			ST D0, \2
13	4i			ENDM
14	10			CodeSec: SECTION
15	11			Start:
16	12			cpChar char1, char2
17	2m	000000	xx xxxx	+ LD D0, char1
18	3m	000003	xx xxxx	+ ST D0, char2
19	13	000006	01	NOP
20	14	000007	01	NOP

Source Listing

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 `LD D0, 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			;-----
3	3			; File: test.o
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			LD D0, \1
12	3i			ST D0, \2
13	4i			ENDM
14	10			CodeSec: SECTION
15	11			Start:
16	12			cpChar char1, char2
17	2m	000000	xx xxxxx +	LD D0, char1
18	3m	000003	xx xxxxx +	ST D0, char2
19	13	000006	01	NOP
20	14	000007	01	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 `LD D0, char1` is located at offset `0` from the section `codeSec` start address.

The instruction `ST D0, 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		LD D0, \1
12	3i		ST D0, \2
13	4i		ENDM
14	10		CodeSec: SECTION
15	11		Start:
16	12		cpChar char1, char2
17	2m	000000 xx xxxx	LD D0, char1
18	3m	000003 xx xxxx	ST D0, char2
19	13	000006 01	NOP
20	14	000007 01	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

Source Listing

```

8      8      000001      char2: DS.B 1
9      9
10     1i
11     2i      INCLUDE "macro.inc"
12     3i      cpChar: MACRO
13     4i          LD D0, \1
14     10          ST D0, \2
15     11          ENDM
16     12          CodeSec: SECTION
17     2m 000000 xx xxxx +          Start:
18     3m 000003 xx xxxx +          cpChar char1, char2
19     13 000006 01          LD D0, char1
20     14 000007 01          ST D0, char2
                          NOP
                          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:

- [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 Parameter Passing Scheme

Check the backend chapter in the compiler manual for the details of parameter passing.

12.2 Return Value

Check the backend chapter in the compiler manual for the details of parameter passing.

12.3 Accessing Assembly Variables in 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;
```

12.4 Accessing ANSI-C Variables in Assembly 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

```
LD D6,CData
....

LD D7,CConst
....
```

NOTE

The compiler supports also the automatic generation of assembler include files. See the description of the `-La` compiler option in the compiler manual.

12.5 Invoking Assembly Function in ANSI-C Source File

An function implemented in an assembly source file (`mixasm.asm` in the following listing) 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 D2, CData ; add CData to the parameter in register A

    ST D2, 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 following listing) 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.

12.5.1 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

```

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

    RAM          = READ_WRITE  0x001000 TO 0x001FFF;
/* EEPROM */
    EEPROM       = READ_ONLY   0x100000 TO 0x100FFF;
/* non -paged FLASHs */
    ROM          = READ_ONLY   0xFF0000 TO 0xFFFFDFF;

END

PLACEMENT

    SSTACK,DEFAULT_RAM INTO RAM;

    DEFAULT_ROM INTO ROM;

END
    
```

```
INIT main
```

12.6 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.
- Define a structured variable.
- Declare a structured variable.
- Access the address of a field inside of a structured variable.
- Access the offset of a field inside of a structured variable.

NOTE

Some limitations apply in the usage of the structured types in the Macro Assembler. See [Structured type: Limitations](#).

The topics covered here:

- [Structured Type Definition](#)
- [Types Allowed for Structured Type Fields](#)
- [Variable Definition](#)
- [Variable Declaration](#)
- [Accessing Structured Variable](#)

12.6.1 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

CodeWarrior Development Studio for Microcontrollers V10.x S12Z Assembler Reference Manual, Rev.
10.6, 02/2014

```

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.

12.6.2 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 12-1. Converting ANSI-C standard types to assembler notation

ANSI-C type	Assembler Notation
char	DS - Define Space
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 --

12.6.3 Variable Definition

The Macro Assembler can provide a way to define a variable with a specific type. This is done using the following syntax:

```
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'
```

12.6.4 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`.

`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'
```

12.6.5 Accessing Structured Variable

The Macro Assembler can provide a means to access each structured type field absolute address and offset.

12.6.5.1 Accessing Field Address

To access a structured-type field address (refer to the listing listed below), 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
```

Support for Structured Types

```

field4:    DS.B  3
field5:    DS.W  1

        ENDSTRUCT

        XREF  myData:myType

        XDEF  entry

CodeSec: SECTION

entry:

        LD D2, myData:field3 ;Loads register D2 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.

12.6.5.2 Accessing 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

entry:

```

```
LD X, #myData
LD D6, (myType->field3,X); Adds the offset of field 'field3'
                           ; (4) to X and loads D6 with the
                           ; content of the pointed address
```

12.7 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 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](#)

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 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
cst2:  DC.B   $BC
```

working with Absolute Sections

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:

LD D0,    cst1 ; Load value in cst1
ADD D0,    cst2 ; Add value in cst2

ST D0,    var  ; Store in var

BRA  entry

```

In the portion of assembly code above, the `LD D0,` 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 A4FA000A00      LD D0,    2560 ; Load value
11     11  a000C05 64FA000A01      ADD D0,    2561 ; Add value
12     12  a000C0A C4FA000800      ST D0,    2561 ; Store in var
13     13  A000C0F 2071              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 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 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:
    LD D0, cst1 ; Load value in cst1
    ADD D0, cst2 ; Add value in cst2
    ST D0, var ; Store in var
    BRA entry
```

In the previous portion of code, the `LD D0,` 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 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 HCS12Z 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

```

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:

    LD D0,  #0
    BRA  int

SWIFunc:

    LD D0,  #4
    BRA  int

ResetFunc:

    LD D0,  #8
    BRA  entry

int:

    PSHH

    LD X,  #Data ; Load address of symbol Data in X
; X <- address of the appropriate element in the tab

Offset:  TST D0,
          TBEQ D0,  Offset3

Offset2:

    INC X  #$1
    DEC A
    BNE Offset2

Offset3:

    INC.W (0,X) ; The table element is incremented
    PULH
    RTI

entry:

    LD S,  #0x10FF; Init Stack Pointer to $1100-$1=$10FF
    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 S12Z 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. 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 Vector Table in Source File Using 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:

    LD D0, #0

    BRA int

SWIFunc:

    LD D0, #4

    BRA int

ResetFunc:

    LD D0, #8

    BRA entry
    
```

```

DummyFunc:
    RTI

int:
    PSHH
    LD X #Data ; Load address of symbol Data in X
    ; X <- address of the appropriate element in the tab

Offset:
    TSTA
    TBEQ D0, Offset3

Offset2:
    INC X #1
    DEC A
    BNE Offset2

Offset3:
    INC.W (0,X) ; The table element is incremented
    PULH
    RTI

entry:
    LD S, #0x10FF ; Init Stack Pointer to $1100-$1=$10FF
    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 Vector Table in Source File Using 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:
        LD D0,  #0
        BRA  int

SWIFunc:
        LD D0,  #4
        BRA  int

ResetFunc:
        LD D0,  #8
        BRA  entry

DummyFunc:
        RTI

int:
        PSHH
        LD X  #Data    ; Load address of symbol Data in X
        ; X <- address of the appropriate element in the tab

Offset:  TSTA
        TBEQ D0,  Offset3

Offset2:
        INC X
        DEC A
        BNE Offset2

Offset3:
        INC.W (0,X); The table element is incremented
        PULH
        RTI

entry:
        LD S,  #0x10FF ; Init Stack Pointer to $1100-$1=$10FF
        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 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 D0, Source
        ST D0, 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 D0 register. The result of the
; computation is stored in the Source variable.
;
; Input Parameter:  The D0 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

          LD D0,   #$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 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

```

Splitting Application into Modules

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.

Chapter 15 Appendices

This document has the following appendices:

- [Global Configuration File Entries](#)
- [Local Configuration File Entries](#)
- [MASM Compatibility](#)
- [MCUasm Compatibility](#)

Chapter 16 Symbols and Notation

This chapter describes the expressions of instruction that represents the variable content such as register names, program labels, and expressions.

The following table lists the symbols & notations with their descriptions:

Table 16-1. Expressions of Instruction

Expression	Description
bwplbwpl	Any of the characters B, W, P, L, or 2-letter pairs BB, BW, BP, BL, WB, WW... LB, LW, LP, or LL to indicate the sizes for an instruction with two input operands. B=byte, W=16-bit word, P=24-bit pointer, L=32-bit long-word. The two-letter codes allow the size of each operand to be specified separately and the one-letter codes indicate the same size is used for both input operands.
bwl	Any of the characters B, W, or L to indicate the size of the operation. B=byte, W=16-bit word, L=32-bit long-word
bwpl	Any of the characters B, W, P, or L to indicate the size of the operation. B=byte, W=16-bit word, P=24-bit pointer, L=32-bit long-word
cc	Branching condition (EQ, NE, MI, PL, GT, or LE) for loop instructions test-and branch (TBcc) or decrement and branch (DBcc). Branch if... EQ - equal; NE - not equal; MI - minus; PL - plus; GT - greater than; LE - less than or equal
cpureg	Any of the CPU registers D0, D1, D2, D3, D4, D5, D6, D7, X, Y, SP, CCH, CCL, or CCW. Used for transfer and exchange instructions.
Di	Any of the eight CPU data registers D2, D3, D4, D5, D0, D1, D6, or D7.
Dj	Typically used for a second operand.
Dk	Used for a third operand in MAC, MOD, MUL, and DIV instructions.
Ds	Used for a source operand.
Dd	Used for a destination operand.
Dn	Used for a numeric control parameter such as the number of positions to shift.

Table continues on the next page...

Table 16-1. Expressions of Instruction (continued)

Expression	Description
Dp	Any of the four 16-bit CPU data registers D2, D3, D4, or D5. Used to specify the width and offset parameters in bit field instructions BFEXT and BFINS.
opr1i opr5i	Any label or expression that evaluates to a 1-bit (5-bit) immediate operand. Used to specify number of shifts for shift and rotate instructions. Immediate value is encoded in the shift postbytes (sb) or (sb+xb).
opr8i	Any label or expression that evaluates to an 8-bit immediate operand.
opr16i	Any label or expression that evaluates to a 16-bit immediate operand.
opr18i	Any label or expression that evaluates to an 18-bit immediate operand. Two bits of the 18-bit operand are encoded into the opcode. The value is zero-extended and placed in X or Y.
opr24	A 24-bit address which can be considered signed or unsigned.
opr24a	A 24-bit address.
opr24i	A 24-bit immediate constant.
opr24u	A 24-bit unsigned constant offset.
opr32i	Any label or expression that evaluates to a 32-bit immediate operand.
oprdest	Any label or expression that evaluates to an address within +127/-128 or +/-16K from the current location. Used for 7-bit or 15-bit relative branches.
oprimsz	Any label or expression that evaluates to an immediate operand of the same size as the CPU register involved in the instruction (8, 16, or 32 bits).
Operators	
oprmemreg	Refer to the <i>CPUS12Z Reference Manual</i> for the OPR addressing summary to see how to expand this into the operand specification for 1 of 16 OPR addressing modes (allowed forms and brief description shown here below).
#oprsxe4i	Short Immediate. oprsxe4i is any label or expression which evaluates to one of the values -1, 1, 2, 3...14, or 15. Auto sign-extended to 8, 16, 24, or 32 bits.
Di	Register as operand. Di is any one of the eight CPU data registers D0, D1, D2, D3, D4, D5, D6, or D7.
(opru4, xys)	Short offset (0-15) from X, Y, or S. opru4 is any label or expression that evaluates to unsigned 0-15.
(+xy) (xy+) (-xy) (xy-) (-S) (S+)	Auto pre/post inc/dec from X, Y, or S (S=SP). Where xy is either of the two index register names X or Y.
(Di, xys)	Register offset from X, Y, or S. xys is any one of the 24-bit indexing registers X, Y, or S (S=SP). 16-bit D2, D3, D4, D5 treated as signed, D0, D1, D6, D7 treated as unsigned.
[Di, xy]	Register offset from X or Y Indirect. D2, D3, D4, D5 treated as signed, D0, D1, D6, D7 are unsigned.

Table continues on the next page...

Table 16-1. Expressions of Instruction (continued)

Expression	Description
(oprs9, xysp)	9-bit signed offset from X, Y, S, or P. oprs9 is any label or expression that evaluates to a 9-bit signed value from -256 to +256. (0 is treated as +256) xysp is any one of the 24-bit registers X, Y, S or P (S=SPP=PC).
[oprs9, xysp]	9-bit signed offset from X, Y, S, or P Indirect.
opru14	Short Extended (16K).

Chapter 17

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`.

The `mcutools.ini` can contain these sections:

- [\[Installation\] Section](#)
- [\[Options\] Section](#)
- [\[XXX_Assembler\] Section](#)
- [\[Editor\] Section](#)

17.1 [Installation] Section

This section lists the entries that can appear in the `[Installation] Section` section of the `mcutools.ini` file. The topics covered here are as follows:

- [Path](#)
- [Group](#)

17.1.1 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
```

17.1.2 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
```

17.2 [Options] Section

This section lists the entries that can appear in the [Options] Section section of the `mcutools.ini` file. The topics covered here are as follows:

- [DefaultDir](#)

17.2.1 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
```

17.3 [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 S12Z Assembler, the name of this section would be [HCS12Z_Assembler].

The topics covered here are as follows:

- [SaveOnExit](#)
- [SaveAppearance](#)
- [SaveEditor](#)
- [SaveOptions](#)
- [RecentProject0, RecentProject1](#)

17.3.1 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.

17.3.2 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.

17.3.3 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](#).

17.3.4 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](#).

17.3.5 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
```

17.4 [Editor] Section

This section lists the entries that can appear in the `[Editor]` section of the `mcutools.ini` file. The topics covered here are as follows:

- [Editor_Name](#)
- [Editor_Exe](#)
- [Editor_Opts](#)

17.4.1 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.

17.4.2 Editor_Exe

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.

17.4.3 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%1,%c
```

17.5 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

[HCS12Z_Assembler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```


Chapter 18

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](#)

18.1 [Editor] Section

This section lists the entries that can appear in the `[Editor] Section` section of the `project.ini` file, where `project` is a placeholder for the name of your project.. The topics covered here are as follows:

- [Editor_Name](#)
- [Editor_Exe](#)
- [Editor_Opts](#)

18.1.1 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.

18.1.2 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.

18.1.3 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
```

18.2 [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 HCS12Z Assembler, the name of this section would be

```
[HCS12Z_Assembler].
```

The topics covered here are as follows:

- [RecentCommandLineX](#), X= integer
- [CurrentCommandLine](#)
- [StatusBarEnabled](#)
- [ToolbarEnabled](#)
- [WindowPos](#)
- [WindowFont](#)
- [TipFilePos](#)
- [ShowTipOfDay](#)
- [Options](#)
- [EditorType](#)
- [EditorCommandLine](#)
- [EditorDDEClientName](#)

- [EditorDDETopicName](#)
- [EditorDDEServiceName](#)

18.2.1 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.

18.2.2 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.

18.2.3 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.

18.2.4 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.

18.2.5 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.

18.2.6 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
```

18.2.7 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.

18.2.8 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.

18.2.9 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.

18.2.10 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.

18.2.11 EditorCommandLine

Arguments

Command line, for UltraEdit-32: " `c:\Programs Files\IDM Software Solutions\UltraEdit-32\uedit32.exe %f -g%1,%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.

18.2.12 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.

18.2.13 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.

18.2.14 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.

18.3 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_Exe=c:\Freescale\prog\idf.exe

Editor_Opts=%f -g%l,%c

[HCS12Z_Assembler]

StatusBarEnabled=1

ToolBarEnabled=1

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%l,%c
```

Chapter 19

MASM Compatibility

The Macro Assembler has been extended to ensure compatibility with the MASM Assembler.

19.1 Comment Line

A line starting with a (*) character is considered to be a comment line by the Assembler.

19.2 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

```

19.3 Operators

For compatibility with the MASM Assembler, the notations listed in the following table are also supported for operators:

Table 19-1. Operator notation for MASM compatibility

Operator	Notation
Shift left	!<
Shift right	!>
Arithmetic AND	!.
Arithmetic OR	!+
Arithmetic XOR	!x, !X

19.3.1 Directives

The following table enumerates the directives that are supported by the Macro Assembler for compatibility with MASM:

Table 19-2. 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 19-2. 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 20

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](#)

20.1 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.

20.2 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.

20.3 Obsolete directives

The following table enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.

Table 20-1. 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.

Index

- "err.log" [171](#)
- (-Fh, [137](#))
- (PC [96](#))
- (usually [96](#))
- (-WmsgFiv, [179](#))
- ArgFile: [126](#)
- AsmDbg [127](#)
- Ci [127](#)
- CMacAngBrack: [128](#)
- CMacBrackets: [129](#)
- Compat [130](#)
- D: [133](#)
- DefLabel [135](#)
- Env: [136](#)
- F2, [137](#)
- F2o, [137](#)
- FA2): [137](#)
- FA2o, [137](#)
- H: [138](#)
- I: [139](#)
- L: [140](#)
- Lasmc: [142](#)
- Lasms: [144](#)
- Lc: [146](#)
- Ld: [148](#)
- Le: [150](#)
- Li: [152](#)
- Lic: [154](#)
- LicA: [155](#)
- LicBorrow: [156](#)
- LicWait: [158](#)
- Ll: [159](#)
- MacroNest: [160](#)
- MCUasm: [160](#)
- N [161](#)
- NoBeep [162](#)
- NoDebugInfo: [163](#)
- NoEnv: [163](#)
- ObjN: [164](#)
- Prod: [165](#)
- Struct: [166](#)
- V: [167](#)
- View: [168](#)
- W1: [169](#)
- W2: [170](#)
- WErrFile: [171](#)
- Wmsg8x3: [172](#)
- WmsgCE: [173](#)
- WmsgCF: [173](#)
- WmsgCI: [174](#)
- WmsgCU: [175](#)
- WmsgCW: [176](#)
- WmsgFb [177](#)
- WmsgFbm [177](#)
- WmsgFbv [177](#)
- WmsgFi [179](#)
- WmsgFim): [179](#)
- WmsgFob: [180](#)
- WmsgFoi: [182](#)
- WmsgFonf: [184](#)
- WmsgFonp: [185](#)
- WmsgNe: [187](#)
- WmsgNi: [188](#)
- WmsgNu: [189](#)
- WmsgNw: [190](#)
- WmsgSd: [191](#)
- WmsgSe: [192](#)
- WmsgSi: [193](#)
- WmsgSw: [193](#)
- WOutFile: [194](#)
- WStdout: [195](#)

- 18-Bit [223](#)
- 24-Bit [221](#), [223](#), [224](#)
- 4-Bit [220](#)
- 8.3 [172](#)
- 9-Bit [221](#), [224](#)

- A**
- A1004 [160](#)
- Abs [306](#)
- ABSENTRY [249](#)
- Absolute [100](#), [199](#), [240](#)
- absolute file [323](#)
- Absolute files [114](#)
- absolute section [339](#)
- absolute sections [204](#), [327](#), [329](#)
- absolute symbols [284](#)
- ABSPATH: [100](#)
- Accompanying [20](#)
- Addition [229](#)
- address [144](#)
- Address [224](#)
- Addressing [218](#), [220](#)
- Addressing Mode [218](#), [219](#), [224](#)
- Addressing Modes [217](#), [219](#), [220](#), [223](#)
- ALIGN [250](#)
- alignment [275](#)
- Analysis [30](#)
- Angle [128](#)
- ANSI-C [312](#), [313](#)
- Appendices [345](#)
- application [329](#), [331](#), [341](#)
- Application [53](#), [168](#), [249](#)

- applications [323](#)
- Applications [323](#)
- area [202](#)
- areas [203](#)
- argument [299](#)
- arguments [129](#)
- Arguments [128](#)
- ASMOPTIONS: [100](#)
- Assemble [89](#)
- Assemble Entry [90](#)
- assembler [100](#)
- Assembler [19](#), [23](#), [40](#), [41](#), [60–66](#), [69](#), [70](#), [73](#), [75](#), [119](#), [121](#), [124](#), [167](#), [207](#), [245](#), [305](#)
- Assembler Applications [311](#)
- assembler directives [249](#)
- Assembler Options [119](#), [126](#)
- Assembler window [116](#)
- Assembler Window [91](#)
- Assembling [39](#), [40](#), [46](#), [89](#)
- assembly [269](#), [271](#), [312](#), [327](#)
- Assembly [36](#), [46](#), [323](#)
- assembly applications [324](#)
- assembly block [268](#)
- Assembly-Control [247](#)
- Assembly directives [248](#)
- assembly file [343](#)
- Assembly file [342](#)
- assembly function [313](#)
- Assembly Language [25](#)
- assembly source file [127](#)
- assembly variables [312](#)
- Associated [87](#)
- attributes [197](#)
- Automatic [222](#)
- available [158](#)

B

- Bar [73](#)
- BASE [251](#)
- batch [180](#)
- Batch mode [116](#)
- beep [162](#)
- Begin [276](#)
- binary [229–232](#), [234](#)
- Binary [242](#)
- Bitwise [232](#), [233](#)
- Blank Lines [293](#)
- Borrow [156](#)
- Box [85](#), [88](#)
- brackets [128](#), [129](#)
- Build Properties [60](#)

C

- call [146](#)
- Calling [298](#)

- C file [314](#)
- Class [87](#)
- CLIST [252](#)
- Code Generation [64](#)
- color [173–176](#)
- COM [79](#)
- command [126](#)
- Command [89](#)
- Command Line [77](#), [78](#), [91](#)
- Comment [207](#)
- Comment Line [369](#)
- compatibility [160](#)
- Compatibility [130](#), [369](#), [373](#)
- Compatibility Modes [64](#)
- Conditional [248](#), [269](#), [271](#)
- conditional assembly [252](#), [261](#)
- Conditional assembly [258](#)
- configuration [96](#)
- Configuration [81](#), [83](#)
- Configure [142](#), [144](#), [160](#)
- Configure Listing File [62](#)
- Configuring [41](#), [78](#)
- Constant [198](#), [220](#), [221](#), [223](#), [224](#)
- Constant Block [256](#)
- Constant-Definition [246](#)
- constants [228](#), [229](#), [288](#)
- Constants [227](#), [369](#)
- continuation [98](#)
- Control directives [247](#), [248](#)
- Copyright [101](#)
- COPYRIGHT: [101](#)
- Create [25](#), [171](#), [194](#)
- Creation [106](#)
- current [102](#)
- Current [94](#)
- CurrentCommandLine [362](#)
- Cut [172](#)

D

- Data [198](#)
- Data-Allocation [246](#)
- DC [254](#)
- DCB [256](#)
- DDE [78](#)
- debug [163](#)
- Debug [115](#)
- debug sections [127](#)
- declaration [318](#)
- Declare [290](#)
- Decrement [222](#)
- Default [100](#), [102](#)
- DefaultDir [352](#)
- DEFAULTDIR: [102](#)
- Define [133](#), [256](#)
- Define Constant [254](#)
- Define Space [257](#)

Defining [202, 203, 297](#)
definition [148, 316, 318](#)
Definition [294](#)
Deleted file [33](#)
details [99](#)
developers [205](#)
development [205](#)
directive [373](#)
Directive [216, 245](#)
directives [246, 247, 374](#)
Directives [245, 370](#)
directory [94, 102, 109, 155](#)
Directory [24](#)
Direct page [296](#)
disable [191](#)
Disable [189](#)
Disable Listing [282](#)
Disable Paging [283](#)
Disable User Messages [66](#)
Display notify box [161](#)
division [230](#)
Document [19](#)
Documentation [20](#)
Drag and Drop [90](#)
DS [257](#)

E

Editor [25, 34, 77, 78, 91, 355, 359](#)
Editor_Exec [356, 360](#)
Editor_Name [355, 359](#)
Editor_Opts [356, 360](#)
EditorCommandLine [366](#)
EditorDDEClientName [366](#)
EditorDDEServiceName [367](#)
EditorDDETopicName [367](#)
Editor Setting [75](#)
EditorType [365](#)
ELF/DWARF [163](#)
ELSE [258](#)
Enable Listing [273](#)
End [261, 262](#)
END [260](#)
End assembly [260](#)
ENDFOR [261](#)
ENDIF [261](#)
ENDM [262](#)
Enhanced [205](#)
entry [101](#)
entry point [249](#)
environment [136, 163](#)
Environment [83, 93, 95, 99, 102](#)
ENVIRONMENT: [102](#)
EQU [263](#)
Equate [263](#)
error [162](#)
Error [90, 116](#)

ERRORFILE: [103](#)
Error message [264](#)
EVEN [263](#)
every [155](#)
Example [87, 240](#)
expansion [150, 302](#)
expression [240, 241](#)
Expression [239](#)
EXT1 [220](#)
EXT2 [220](#)
EXT24 [220](#)
EXT3 [220, 224](#)
Extended [220](#)
External [25, 226](#)
External Reference [296](#)
External Symbol [294, 295](#)

F

FAIL [264](#)
fatal [173](#)
feature [155, 156](#)
Feedback [90](#)
field [208, 209](#)
field address [319](#)
field offset [320](#)
File Menu [73](#)
filename [164](#)
Filename [103](#)
filenames [172](#)
files [113, 114, 163](#)
Files [30, 113](#)
floating [158](#)
Floating-Point [229](#)
FOR [268](#)
FOR block [261](#)
Force [237, 263](#)
Forcing [275](#)
format [137, 172, 179, 180, 182, 184, 185](#)

G

General [66](#)
Generate [140](#)
generating [323](#)
GENPATH: [105](#)
Global [96](#)
Global Configuration [351](#)
Global Editor [76](#)
Graphical User Interface [69](#)
Group [33, 352](#)
grouping [128, 129, 299](#)
Groups [30, 31](#)

H

HCS12Z [209](#)

header [306](#)
 Help [138](#)
 HIGH [235](#)
 HIGH_6_13 [236](#)
 Highlights [19](#)
 Host [64](#)

I

IDE [25, 39, 53](#)
 IDX [220, 222, 223](#)
 IDX1 [221, 224](#)
 IDX2 [223](#)
 IDX3 [221, 223, 224](#)
 IF [269](#)
 IFcc [271](#)
 IMMe4 [219](#)
 Immediate [218, 219](#)
 Include [113, 139](#)
 INCLUDE [272](#)
 included [152](#)
 include file [342](#)
 INCLUDETIME: [106](#)
 Increment [222](#)
 Indexed [220, 222–224](#)
 Indirect [223, 224](#)
 Inherent [218](#)
 initialization [96](#)
 input [105](#)
 Input [63, 89, 113](#)
 Input Files [45](#)
 Insert [293](#)
 Installation [351](#)
 Instruction set [209](#)
 Instruction Set [209](#)
 integer [362](#)
 Integer [228](#)
 Integers [369](#)
 interactive [179, 182](#)
 Interactive mode [116](#)
 Invoking [313](#)

L

label [159](#)
 Label [133, 208](#)
 Labels [301, 373](#)
 Language [63, 64](#)
 license [156, 158](#)
 License [154, 155, 158](#)
 Limitations [321](#)
 limits [242](#)
 Line Length [274](#)
 Line Number [91](#)
 linker [333](#)
 Linker [56](#)
 Linking [329, 331](#)

List [124, 252, 279](#)
 LIST [273](#)
 listing [140, 142, 144, 146, 148, 150, 152, 194](#)
 Listing [115, 305](#)
 Listing-File [247](#)
 listing files [115](#)
 Listing Files [35](#)
 Listing Title [294](#)
 LLEN [274](#)
 Loc [308](#)
 Local [96](#)
 Local Configuration [359](#)
 Local Editor [76](#)
 Location Counter [250, 286](#)
 Logical [233](#)
 LONGEVEN [275](#)
 Long-Word [275](#)
 LOW [236](#)

M

macro [129, 148, 160, 297](#)
 Macro [128, 146, 150, 217, 248, 297–299, 302](#)
 MACRO [276](#)
 macro definition [262, 276](#)
 Macro Expansion [277](#)
 macro expansions [279](#)
 macros [95, 298, 301, 302](#)
 Macros [297](#)
 Main Window [70](#)
 Make [323](#)
 MAP_ADDR_6 [237](#)
 MASM [369](#)
 maximum [160](#)
 mctools.ini [96](#)
 MCUasm [160, 373](#)
 Memory maps [324](#)
 Menu [75](#)
 Menu Bar [73](#)
 Message [90](#)
 messages [169, 170, 173–176, 187–190](#)
 Messages [65, 66](#)
 MEXIT [277](#)
 Microsoft [172](#)
 Mixed C [311, 324](#)
 MLIST [279](#)
 mode [179, 180, 182](#)
 Mode [218](#)
 Modes [220](#)
 modifiers [121](#)
 Modifiers [80](#)
 Modularity [204](#)
 modules [341](#)
 modulo [230](#)
 multiple [203](#)
 Multiple [205](#)
 Multiplication [230](#)

N

Name [110](#)
Nested [302](#)
nesting [160](#)
New [25](#)
New File [32](#)
New Group [32](#)
NOLIST [282](#)
NOPAGE [283](#)
not [163](#)
Notation [347](#)
Number [187](#), [188](#), [190](#)
number base [251](#)

O

Obj [309](#)
object [101](#), [106](#), [110](#)
Object [107](#), [114](#), [164](#)
Object-code [55](#)
object files [323](#)
OBJPATH: [107](#)
Obsolete [374](#)
occurrence [168](#)
Offset [220–224](#)
OFFSET [284](#)
Online Help [160](#)
only) [96](#)
Operand Field [217](#)
operation [241](#)
Operation [209](#)
operations [242](#)
operator [235–237](#)
Operator [236](#), [237](#), [239](#)
operators [229–234](#)
Operators [229](#), [370](#)
Option [84](#)
Option Details [121](#)
options [100](#), [126](#)
Options [119](#), [124](#), [352](#), [365](#)
ORG [286](#)
output [195](#)
Output [61](#), [62](#), [114](#), [137](#)
overlaps [206](#)
overview [297](#)

P

PAGE [287](#)
Page break [287](#)
Page Length [288](#)
Panels [60](#)
Parameter [311](#)
parameters [298](#)
passing scheme [311](#)
path [100](#), [105](#), [107](#), [109](#), [139](#)

Path [351](#)
PLEN [288](#)
portability [205](#)
position [185](#)
Post [222](#)
Pre [222](#)
precedence [239](#)
Prints [167](#)
PRM [53](#), [333](#), [343](#)
processing [117](#)
project.ini) [96](#)
Project Files [37](#)
projects [76](#)

R

RAD50 [288](#)
RAD50-encoded [288](#)
RAM [202](#), [203](#)
read [126](#)
RecentCommandLineX [362](#)
RecentProject0 [354](#)
RecentProject1 [354](#)
Reference [295](#)
REG [222](#), [223](#)
Register [218](#), [222](#), [223](#)
Rel [307](#)
REL [219](#)
REL1 [219](#)
Relational [234](#)
Relative [219](#)
relocatable [241](#)
Relocatable [201](#), [204](#)
relocatable section [336](#)
Relocatable Section [290](#)
relocatable sections [330](#), [331](#)
Removing [33](#)
Renaming [33](#)
Repeat [268](#)
Reserved [227](#)
Restoring [33](#)
result [241](#), [242](#)
Return Value [311](#)
Reusability [206](#)
RGB [173–176](#)
ROM [202](#), [203](#)

S

S12Z [19](#)
S12Z Assembler [61](#)
SaveAppearance [353](#)
SaveEditor [354](#)
SaveOnExit [353](#)
SaveOptions [354](#)
Search [105](#)
Section [197](#)

SECTION 290
Section-Definition 245
sections 197–199, 201
Sections 197
Section types 198
segmentation 324
Server 158
Set 136, 179, 288
SET 292, 373
Setting 191–193
Settings 84, 85
shared 76
Shift operators 231
Short 138, 220
Show 159
ShowTipOfDay 365
Sign 231
Simple 241
size 144
Source 207
Source-code Files 46
source file 312, 313, 327, 330, 336, 339
Source files 113
Source Files 36, 39
Source line 309
Source listing 306
SPC 293
special 121
special modifiers 122
specification 102, 103, 164
Specify 126, 165
Specifying 89
Square 129
S-Record 108, 115
SRECORD: 108
standard 168, 195
startup 165
statistics 159
Status 73
StatusBarEnabled 362
string 288
String 228
Structure 19
structured 166
Structured type 316, 321
structured type fields 317
structured types 316
structured variable 319
subtraction 229
Support 166
Switch 160
Symbol-Linkage 246
symbols 225–227
Symbols 225, 296, 347
symbol value 263
Symbol Value 292
Syntax 207

T

Tab Length 293
TABS 293
Temporary 109
Terminate 277
Test1.asm 342
Test1.inc 342
Test2.asm 343
Test2.prm 343
Text 109
TEXTPATH: 109
this 19
time 106
TipFilePos 364
TITLE 294
TMP: 109
Toolbar 72, 89
ToolbarEnabled 363
tools 76
Tracking 206
Translation 242
type 108
types 166

U

unary 231, 233, 237
Unary 241
Undefined 227
use 163
user 175, 189
User 110
USERNAME: 110

V

variable 136
Variable 318
variables 99, 312
Vector table 333
Vector Table 336, 339
version 167
View Menu 75

W

Wait 158
warning 170, 176
Warning 190, 193
will 126
WindowFont 364
WindowPos 363
Window Title 71
word alignment 263
Working 23
Write 195

X

XDEF [294](#)

XREF [295](#)

XREFB [296](#)

XXX_Assembler [353](#), [361](#)



How to Reach Us:

Home Page:

freescale.com

Web Support:

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.

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.