HC(S)12 Assembler Manual

Revised: 9 February 2006

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

Copyright © 2006 by Freescale Semiconductor, Inc. All rights reserved.

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

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

How to Contact Us

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

Table of Contents

I Using the HC(S)12 Assembler

Highlights 15
Structure of this document 15

1 Working with the Assembler 17

Programming Overview 17
Project directory 18
External Editor 18
Using CodeWarrior to manage an assembly language project 19
The New Project Wizard 19
Analysis of groups and files in the project window 31
CodeWarrior groups 32
Writing your assembly source files. 33
Analyzing the project files 33
Assembling your source files 35
Assembling with CodeWarrior 35
Assembling with the Assembler 37
Linking the application 50
Linking with CodeWarrior. 50
Linking with the Linker 54
Directly generating an ABS file 60
Using CodeWarrior to generate an ABS file 60
Using the Assembler for absolute assembly 65

2 Assembler Graphical User Interface 73

Starting the Assembler 73
Assembler main window 74
Window title 75
Content area. 75
Toolbar 77

Status bar 78
Assembler menu bar 78
File menu 78
Assembler menu 80
View menu 80
Editor Settings dialog box 81
Global Editor (shared by all tools and projects) 81
Local Editor (shared by all tools) 82
Editor started with the command line 83
Editor started with DDE 85
CodeWarrior with COM 86
Modifiers 86
Save Configuration dialog box 87
Environment Configuration dialog box 88
Option Settings dialog box 89
Message Settings dialog box 89
Changing the class associated with a message 91
About... dialog box 92
Specifying the input file 92
Use the command line in the toolbar to assemble 92
Use the File > Assemble... entry 93
Use Drag and Drop 93
Message/Error feedback 93
Use information from the assembler window 94
Use a user-defined editor 94

3 Environment 97

Current directory 98
Environment macros 99
Global initialization file - mcutools.ini (PC only) 99
Local configuration file (usually project.ini) 100
Paths 101
Line continuation. 102
ABSPATH: Absolute file path 104
ASMOPTIONS: Default assembler options 105

COPYRIGHT: Copyright entry in object file 106
DEFAULTDIR: Default current directory. 107
ENVIRONMENT: Environment file specification 108
ERRORFILE: Filename specification error 109
GENPATH: Search path for input file 112
INCLUDETIME: Creation time in the object file 113
OBJPATH: Object file path 114
SRECORD: S-Record type 115
TEXTPATH: Text file path 116
TMP: Temporary directory 117
USERNAME: User Name in object file 118

4 Files 119

Input files 119
Source files 119
Include files 119
Output files 119
Object files 120
Absolute files. 120
S-Record Files 120
Listing files 121
Debug listing files 121
Error listing file 121
File Processing 122

5 Assembler Options 123

Types of assembler options 123
Assembler Option details 125
Using special modifiers 125
List of assembler options 128
Detailed listing of all assembler options 131
-C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON 132
-Ci: Switch case sensitivity on label names OFF 133
-CMacAngBrack: Angle brackets for grouping Macro Arguments 135

-CMacBrackets: Square brackets for macro arguments grouping 136
-Compat: Compatibility modes 137
-CpDirect: Define DIRECT register value 140
-Cpu (-CpuCPU12, -CpuHCS12, -CpuHCS12X): Derivative 143
-D: Define Label 146
-Env: Set environment variable 148
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output-file format 149
-H: Short Help 151
-I: Include file path 152
-L: Generate a listing file 153
-Lasmc: Configure listing file 156
-Lasms: Configure the address size in the listing file 159
-Lc: No Macro call in listing file 161
-Ld: No macro definition in listing file 164
-Le: No Macro expansion in listing file 167
-Li: Not included file in listing file 170
-Lic: License information 172
-LicA: License information about every feature in directory 173
-LicBorrow: Borrow license feature 174
-LicWait: Wait until floating license is available from floating
License Server 176
-M (-Ms, -Mb, -Ml): Memory Model 177
-MacroNest: Configure maximum macro nesting 178
-MCUasm: Switch compatibility with MCUasm ON 179
-N: Display notify box 180
-NoBeep: No beep in case of an error 181
-NoDebugInfo: No debug information for ELF/DWARF files 182
-NoEnv: Do not use environment 183
-ObjN: Object filename specification 184
-Prod: Specify project file at startup 186
-Struct: Support for structured types 187
-V: Prints the Assembler version 188
-View: Application standard occurrence 189
-W1: No information messages 191
-W2: No information and warning messages 192

-WErrFile: Create "err.log" error file 193
-Wmsg8x3: Cut filenames in Microsoft format to 8.3 194
-WmsgCE: RGB color for error messages 196
-WmsgCF: RGB color for fatal messages. 197
-WmsgCI: RGB color for information messages 198
-WmsgCU: RGB color for user messages. 199
-WmsgCW: RGB color for warning messages 200
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode 201
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode 204
-WmsgFob: Message format for batch mode 206
-WmsgFoi: Message format for interactive mode. 208
-WmsgFonf: Message format for no file information. 210
-WmsgFonp: Message format for no position information. 212
-WmsgNe: Number of error messages 214
-WmsgNi: Number of Information messages 215
-WmsgNu: Disable user messages 216
-WmsgNw: Number of Warning messages. 218
-WmsgSd: Setting a message to disable 219
-WmsgSe: Setting a message to Error. 220
-WmsgSi: Setting a message to Information. 221
-WmsgSw: Setting a Message to Warning 222
-WOutFile: Create error listing file. 223
-WStdout: Write to standard output 224

6 Sections 225

Section attributes. 225
Code sections. 225
Constant sections. 225
Data sections 226
Section types 226
Absolute sections. 226
Relocatable sections 228
Relocatable vs. absolute sections 231

Modularity 231
Multiple developers 231
Early development. 232
Enhanced portability 232
Tracking overlaps 232
Reusability 232

7 Assembler Syntax 233

Comment line 233
Source line. 233
Label field 234
Operation field. 234
Operand field: Addressing modes 246
Comment field 259
Symbols. 260
User-defined symbols 260
External symbols 261
Undefined symbols 261
Reserved symbols. 262
Constants 262
Integer constants 262
String constants 263
Floating-Point constants 263
Operators 263
Addition and subtraction operators (binary) 263
Multiplication, division and modulo operators (binary) 264
Sign operators (unary) 265
Shift operators (binary) 265
Bitwise operators (binary) 266
Bitwise operators (unary) 267
Logical operators (unary) 267
Relational operators (binary). 268
HIGH operator. 269
PAGE operator. 270
Force operator (unary). 270

Operator precedence 271
Expression. 272
Absolute expression 273
Simple relocatable expression. 274
Unary operation result. 274
Binary operations result 275
Translation limits 276

8 Assembler Directives 277

Directive overview 277
Section-Definition directives. 277
Constant-Definition directives 277
Data-Allocation directives. 278
Symbol-Linkage directives 278
Assembly-Control directives. 278
Listing-File Control directives 279
Macro Control directives. 280
Conditional Assembly directives 280
Detailed descriptions of all assembler directives 281
ABSENTRY - Application entry point 282
ALIGN - Align Location Counter. 284
BASE - Set number base. 285
CLIST - List conditional assembly 286
DC - Define Constant 288
DCB - Define Constant Block. 290
DS - Define Space. 292
ELSE - Conditional assembly 294
END - End assembly. 296
ENDFOR - End of FOR block 297
ENDIF - End conditional assembly 298
ENDM - End macro definition 299
EQU - Equate symbol value 300
EVEN - Force word alignment 301
FAIL - Generate Error message. 303
FOR - Repeat assembly block. 307

IF - Conditional assembly 309
IFcc - Conditional assembly 311
INCLUDE - Include text from another file 313
LIST - Enable Listing 314
LLEN - Set Line Length. 316
LONGEVEN - Forcing Long-Word alignment 318
MACRO - Begin macro definition 319
MEXIT - Terminate Macro Expansion 320
MLIST - List macro expansions 322
NOLIST - Disable Listing 325
NOPAGE - Disable Paging 327
OFFSET - Create absolute symbols 328
ORG - Set Location Counter 330
PAGE - Insert Page break 332
PLEN - Set Page Length 334
RAD50 - Rad50-encoded string constants 335
SECTION - Declare Relocatable Section 338
SET - Set Symbol Value 340
SPC - Insert Blank Lines 341
TABS - Set Tab Length 342
TITLE - Provide Listing Title 343
XDEF - External Symbol Definition. 344
XREF - External Symbol Reference 345
XREFB - External Reference for Symbols located on the Direct Page . . .346

9 Macros 347

Macro overview. 347
Defining a macro 347
Calling macros. 348
Macro parameters 348
Macro argument grouping 349
Labels inside macros 350
Macro expansion 352
Nested macros 352

10 Assembler Listing File 355

Page header 355
Source listing 356
Abs. 356
Rel. 357
Loc. 358
Obj. code 359
Source line 360

11 Mixed C and Assembler Applications 361

Memory models 361
Parameter passing scheme 362
Return Value 363
Accessing assembly variables in an ANSI-C source file 363
Accessing ANSI-C variables in an assembly source file 364
Invoking an assembly function in an ANSI-C source file 365
Example of a C file 366
Support for structured types 368
Structured type definition 368
Types allowed for structured type fields 369
Variable definition 370
Variable declaration 370
Accessing a structured variable 371
Structured type: Limitations 373

12 Make Applications 375

Assembly applications 375
Directly generating an absolute file 375
Mixed C and assembly applications 375
Memory maps and segmentation 376

13 How to ... 377

How to work with absolute sections 377
Defining absolute sections in an assembly source file 377

Linking an application containing absolute sections 379
How to work with relocatable sections 380
Defining relocatable sections in a source file 380
Linking an application containing relocatable sections. 381
How to initialize the Vector table 383
Initializing the Vector table in the linker PRM file 383
Initializing the Vector table in a source file using a relocatable section . . .385
Initializing the Vector table in a source file using an absolute section 388
Splitting an application into different modules 390
Example of an assembly file (Test1.asm) 390
Corresponding include file (Test1.inc) 391
Example of an assembly File (Test2.asm) 391
Using the direct addressing mode to access symbols 392
Using the direct addressing mode to access external symbols 393
Using the direct addressing mode to access exported symbols 393
Defining symbols in the direct page 394
Using the force operator 394
Using SHORT sections 395

II Appendices

A Global Configuration File Entries 399

[Installation] Section 400
Path 400
Group 400
[Options] Section. 401
DefaultDir 401
[XXX_Assembler] Section 402
SaveOnExit 402
SaveAppearance 402
SaveEditor 403
SaveOptions. 403
RecentProject0, RecentProject1, ...403

[Editor] Section 405
Editor_Name 405
Editor_Exe 405
Editor_Opts 406
Example 407

B Local Configuration File Entries 409

[Editor] Section 409
Editor_Name 410
Editor_Exe 411
Editor_Opts 412
[XXX_Assembler] Section 413
RecentCommandLineX, X= integer 414
CurrentCommandLine 415
StatusbarEnabled 416
ToolbarEnabled 417
WindowPos 418
WindowFont 419
TipFilePos 420
ShowTipOfDay 421
Options 422
EditorType 423
EditorCommandLine 424
EditorDDEClientName 425
EditorDDETopicName 426
EditorDDEServiceName 427
Example 428

C MASM Compatibility 429

Comment Line 429
Constants (Integers) 429
Operators 430
Directives 430

D MCUasm Compatibility 433

Labels 433
SET directive.....433
Obsolete directives 434

E Semi-Avocet Compatibility 435

Directives 435
Section Definition 437
Macro parameters 439
Support for Structured Assembly 439
SWITCH block 439
FOR Block.....440

Index 443

I

Using the HC(S)12 Assembler

This document explains how to effectively use the HC(S)12 Macro Assembler.

Highlights

The major features of the HC(S)12 Assembler are:

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

Structure of this document

This section has the following chapters:

- “Working with the Assembler” on page 17: A tutorial for creating assembly-language projects using the CodeWarrior Development Suite or the standalone Build Tools. Both relocatable and absolute assembly projects are created. Also a description of the Assembler’s environment that creates and edits assembly source code and assembles the source code into object code which could be further processed by the Linker.
- “Assembler Graphical User Interface” on page 73: A description of the Macro Assembler’s Graphical User Interface (GUI)
- “Environment” on page 97: A detailed description of the Environment variables used by the Macro Assembler

- “Files” on page 119: A description of the input and output file the Assembles uses or generates.
- “Assembler Options” on page 123: A detailed description of the full set of assembler options
- “Sections” on page 225: A description of the attributes and types of sections
- “Assembler Syntax” on page 233: A detailed description of the input syntax used in assembly input files.
- “Assembler Directives” on page 277: A list of every directive that the Assembler supports
- “Macros” on page 347: A description of how to use macros with the Assembler
- “Assembler Listing File” on page 355: A description of the assembler output files
- “Mixed C and Assembler Applications” on page 361: A description of the important issues to be considered when mixing both assembly and C source files in the same project
- “Make Applications” on page 375: A description of special issues for the linker
- “How to ...” on page 377: Examples of assembly source code, linker PRM, and assembler output listings.

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

- “Global Configuration File Entries” on page 399: Description of the sections and entries that can appear in the global configuration file - `mcutools.ini`
- “Local Configuration File Entries” on page 409: Description of the sections and entries that can appear in the local configuration file - *`project`*`.ini`
- “MASM Compatibility” on page 429: Description of extensions for compatibility with the MASM Assembler
- “MCUasm Compatibility” on page 433: Description of extensions for compatibility with the MCUasm Assembler
- “Semi-Avocet Compatibility” on page 435

1

Working with the Assembler

This chapter is primarily a tutorial for creating and managing HC(S)12 assembly projects with the CodeWarrior Development Studio. In addition, there are instructions to utilize the Assembler and Smart Linker Build Tools in the CodeWarrior Development Studio for assembling and linking assembly projects.

This chapter covers the following topics:

- “Programming Overview” on page 17
- “Using CodeWarrior to manage an assembly language project” on page 19
- “Analysis of groups and files in the project window” on page 31
- “Writing your assembly source files” on page 33
- “Analyzing the project files” on page 33
- “Assembling your source files” on page 35
- “Linking the application” on page 50
- “Directly generating an ABS file” on page 60
- “Using the Assembler for absolute assembly” on page 65

Programming Overview

In general terms, an embedded systems developer programs small but powerful microprocessors to perform specific tasks. These software programs for controlling the hardware is often referred to as firmware. One such end use for firmware might be controlling small stepper motors in an automobile seat which “remember” their settings for different drivers or passengers.

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 which are specially tailored to types 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 subfamilies of their embedded microprocessors.

Assembly language instructions are considered as being at a lower level (closer to the hardware) than the essentially standardized C instructions. 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 routines are usually faster to execute than their C counterparts, but may require much more programming effort. Therefore, assembly-language programming is usually considered only for those critical applications which

take advantage of its higher speed. 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 like C use compilers 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 on page 361 chapter.

This manual covers the Assembler designed for the Freescale 16-bit HC(S)12 series of microcontrollers. There is a companion manual for this series that covers the HC(S)12 Compiler.

The HC(S)12 Assembler can be used as a transparent, integral part of the CodeWarrior Development Studio. 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 one of the Build Tool Utilities included with CodeWarrior such as a Linker, Compiler, ROM Burner, Simulator or Debugger, etc.

The typical configuration of an Assembler (or any of the other Build Tool Utilities) is its association with a Project directory on page 18 and an External Editor on page 18. The CodeWarrior Development Studio uses a project directory for storing the files it creates and coordinates the various Build Tools. The Assembler is but one of these tools that the CodeWarrior IDE coordinates. The tools used most frequently within CodeWarrior are its integrated Editor, Compiler, Assembler, Linker, the Simulator/Debugger, and Processor Expert. Most of these “Build Tools” are located in the *prog* subfolder of the CodeWarrior installation. The others are directly integrated into the CodeWarrior IDE.

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

Project directory

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

In the process of designing a project, you can either start from scratch by designing your own source-code, configuration (`*.ini`), and various layout files for your project for use with standalone project-building tools. This was how embedded microprocessor projects were developed in the recent past. On the other hand, you can have the CodeWarrior IDE coordinate the Build Tools and transparently manage the entire project. This is recommended because it is far easier and faster than employing standalone tools. However, you can still utilize any of the separate Build Tools in the CodeWarrior Development Studio suite.

External Editor

CodeWarrior reduces programming effort because its internal editor is configured with the Assembler to enable both positive and error feedback. You can use the *Configuration* dialog box of the standalone Assembler or other standalone Build Tools in the CodeWarrior Development Studio to configure or select your editor. Please refer to the Editor Settings dialog box on page 81 section of this manual.

Using CodeWarrior to manage an assembly language project

CodeWarrior has an integrated New Project 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 generate the files that are located in the project directory.

This section will create a basic CodeWarrior project that uses HC(S)12 assembly source code exclusively - no C source code. A sample program is included for a project created using the Wizard. For example, the program included for an assembly project calculates the next number in a mathematical Fibonacci series. It is much easier to analyze any program if you already have some familiarity with solving the result in advance. Therefore, the following paragraph describes a Fibonacci series.

In case you did not know, a Fibonacci series is a mathematical infinite series that is very easy to visualize (Listing 1.1 on page 19):

Listing 1.1 Fibonacci series

```
 0,  1,  1,   2,  3,  5,  8,  13,  21,  34,  55,  89,... to infinity -->
[start] 1st  2nd ...      ... 6th Fibonacci term
```

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

Let's now rapidly create a project with CodeWarrior and analyze the assembly source and the Linker's parameter files to calculate a Fibonacci series for a particular 16-bit microprocessor in the Freescale HC(S)12 family - in this case, the MC9S12C32.

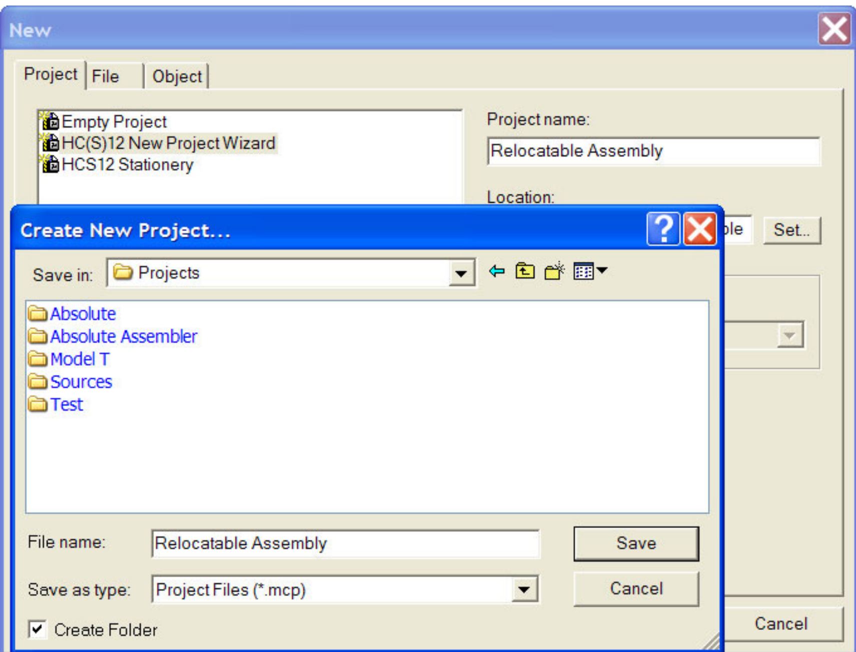
The New Project Wizard

Start the HC(S)12 CodeWarrior IDE application. Its path is:

```
<CodeWarrior installation folder>\bin\IDE.exe
```

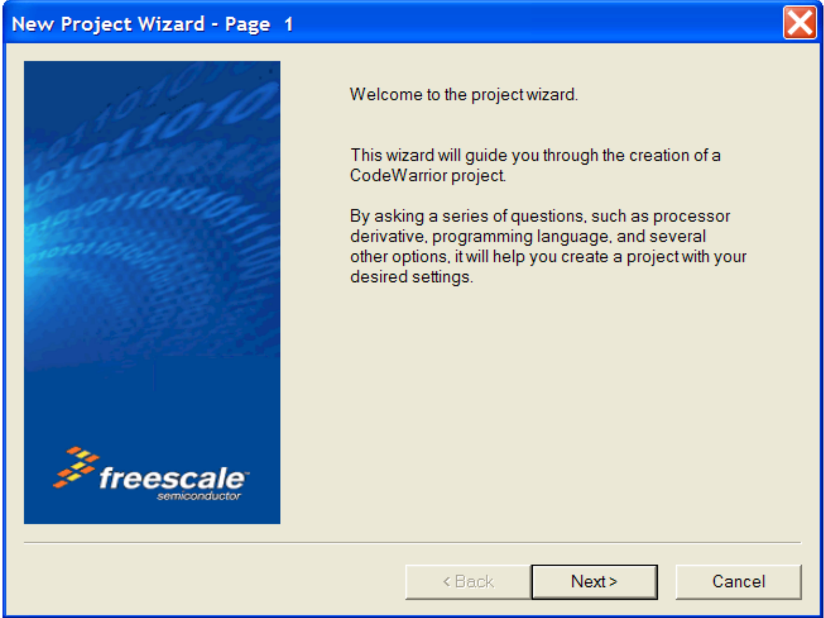
After CodeWarrior opens, close the *Tip of the Day* dialog box (if it opens when CodeWarrior opens) and select from the *File* menu: *File > New...* . The *New* dialog box appears (Figure 1.1 on page 20).

Figure 1.1 New dialog box



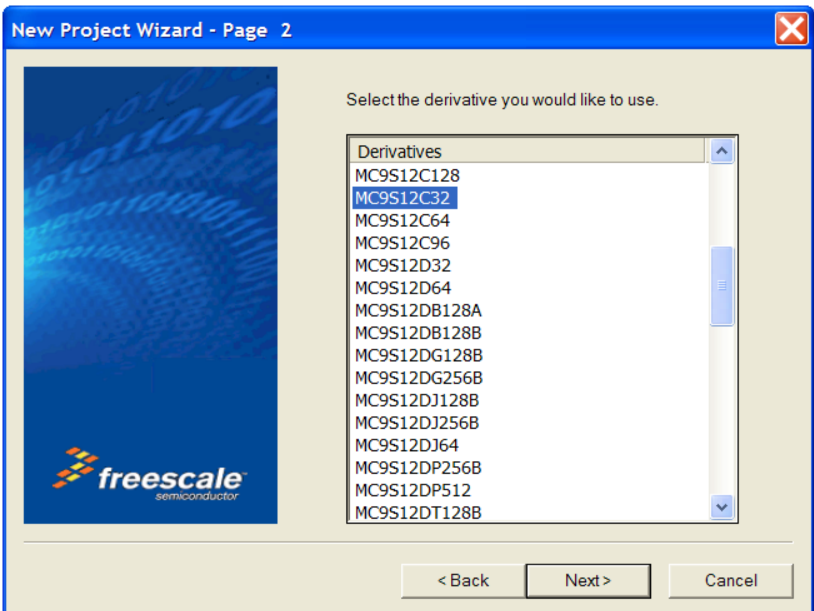
From the default *Project* sheet, select *HC(S)12 New Project Wizard*. Enter a name for your project in the *Project Name:* text box. If you want a different path for your project than the default path displayed in the *Location:* text box, either enter the new path in the text box or press the *Set* button and browse to the new location. Check the *Create Folder* check box, unless you already have an existing project directory. Press *Save* and *OK* to close the dialog boxes. The *New Project Wizard - Page 1* dialog box appears (Figure 1.2 on page 21).

Figure 1.2 New Project Wizard - Page 1 dialog box



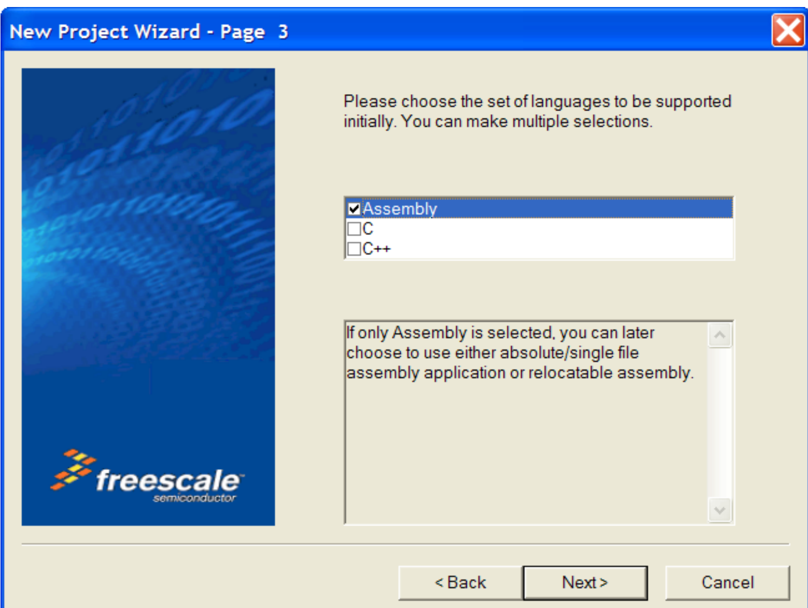
Press *Next* >. The *New Project Wizard - Page 2* dialog box appears (Figure 1.3 on page 22).

Figure 1.3 New Project Wizard - Page 2 dialog box



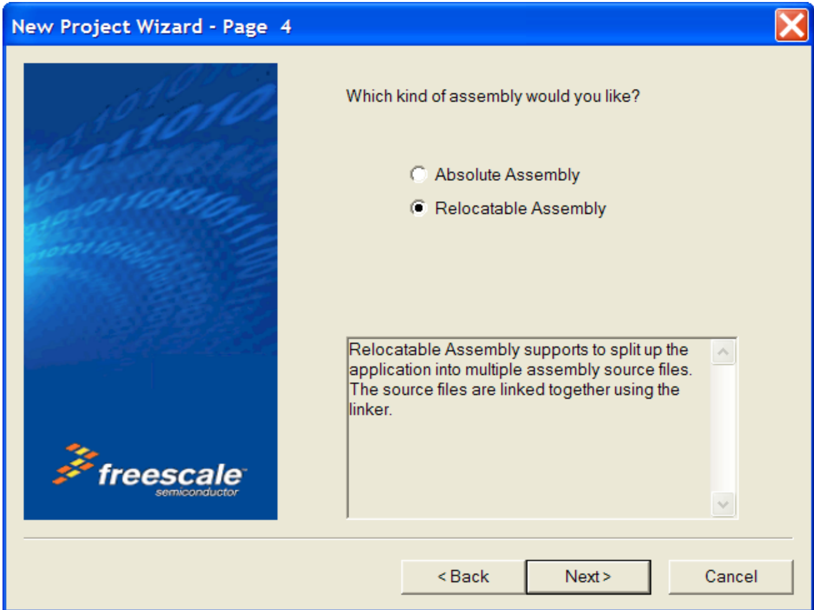
Select the CPU derivative you want for your project. In this case, the *MC9S12C32* is selected. Press *Next* >. The *New Project Wizard - Page 3* dialog box appears (Figure 1.4 on page 23).

Figure 1.4 New Project Wizard - Page 3 dialog box



This is an assembly project, so HC(S)12 Assembly is the language. Check *Assembly* and be sure than *C* and *C++* are unchecked. Press *Next* >. The *New Project Wizard - Page 4* dialog box appears (Figure 1.5 on page 24).

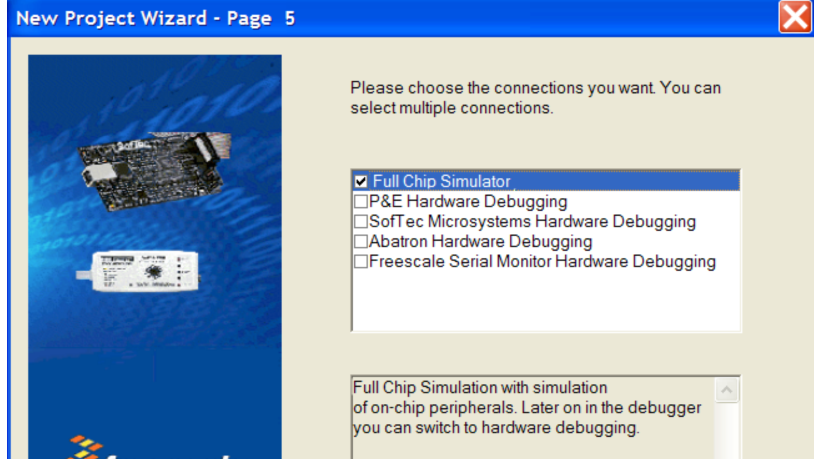
Figure 1.5 New Project Wizard - Page 4 dialog box



Select *Relocatable Assembly*. Although this project will only use one `*.asm` assembly source code file, it is more flexible to use relocatable assembly in case the project expands to include additional assembly source files. Because this project will use only one `*.asm` file, the other option - *Absolute Assembly* - will be covered later in this chapter. The *New Project Wizard - Page 5* dialog box appears (Figure 1.6 on page 25).

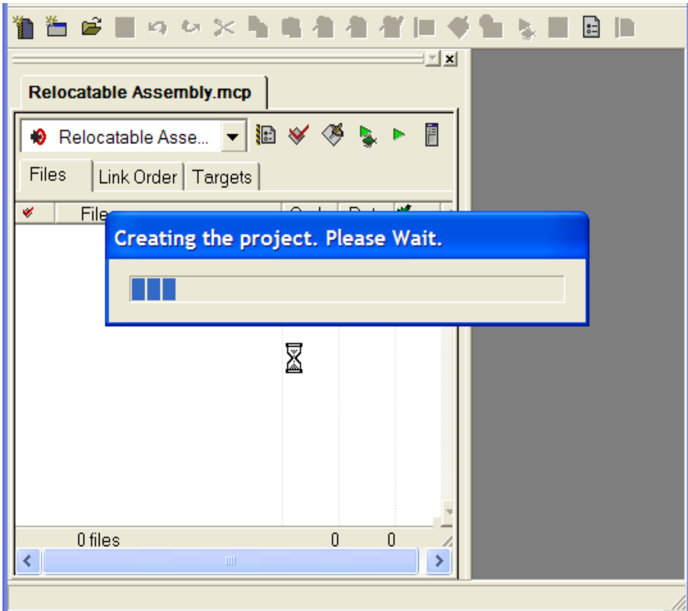
NOTE If an assembly project has two or more assembly source (`*.asm`) files, the *Relocatable Assembly* option must be selected.

Figure 1.6 New Project Wizard - Page 5 dialog box



The default - *Full Chip Simulator* - is selected. Press *Finish* >. The Wizard now creates the project (Figure 1.7 on page 26).

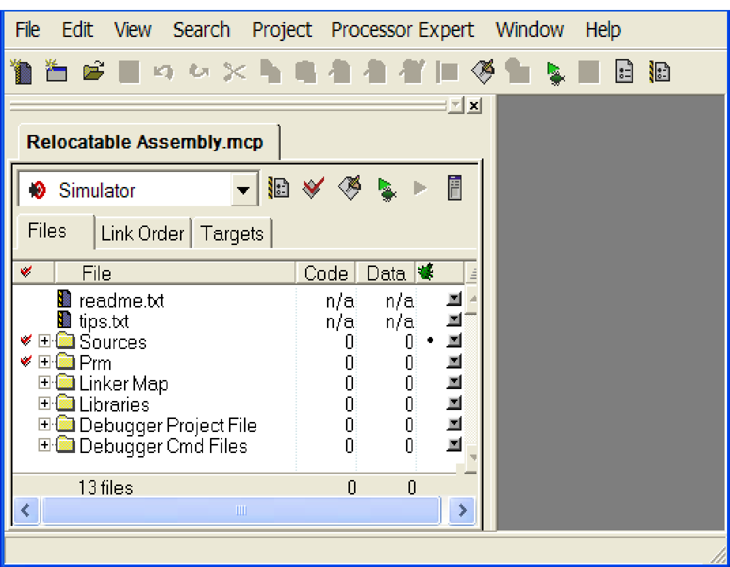
Figure 1.7 The CodeWarrior project is being created...



You can (but do it later) safely close the CodeWarrior IDE at any time after this point, and your project will be automatically configured in its previously-saved status when you work on the project later.

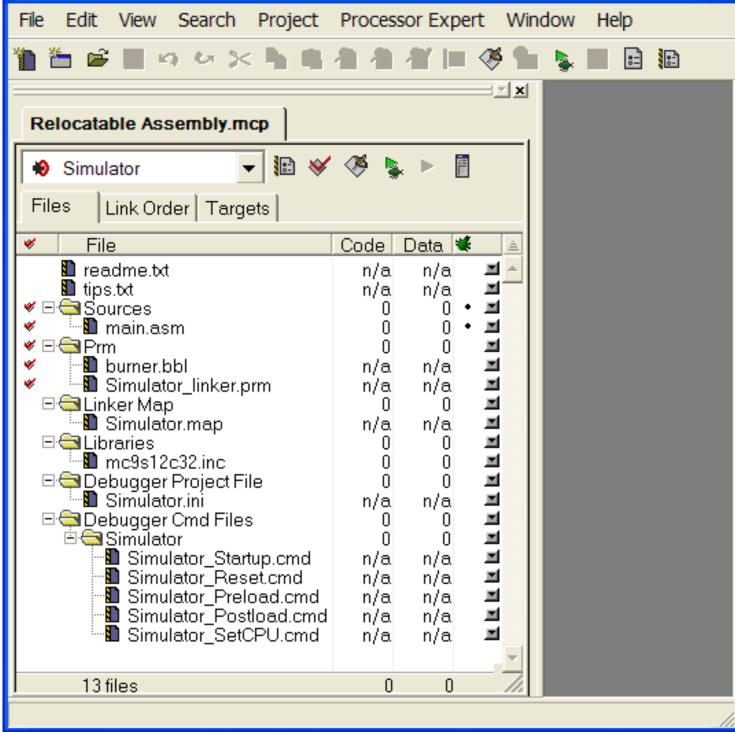
Using the New Project Wizard, an HC(S)12 project is set up in a matter of a minute or two. You can add additional components to your project afterwards. A number of files and folders are automatically generated in the root folder that was used in the project-naming process. This folder is referred to in this manual as the project directory. The CodeWarrior project window appears (Figure 1.8 on page 27).

Figure 1.8 CodeWarrior project window



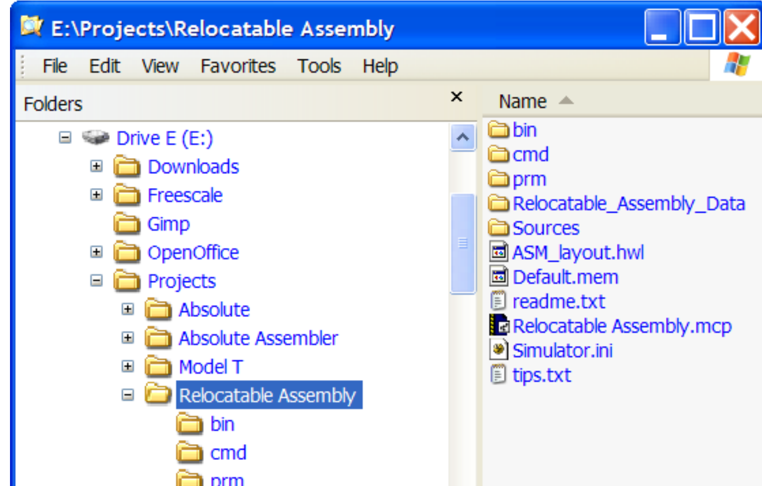
If you expand the six “folder“ icons, actually groups of files, by clicking in the CodeWarrior project window, you could view the files that CodeWarrior generated. In general, any folders or files in the project window with red check marks will remain so checked until the files are successfully assembled, compiled, or linked (Figure 1.9 on page 28).

Figure 1.9 Project window showing the files that CodeWarrior created



You could use the Windows Explorer to examine the actual folders and files that CodeWarrior generated for your project and displays in the project window above, as in Figure 1.10 on page 29. The name and location for the project directory are what you selected when creating the project with the Wizard.

Figure 1.10 Project directory in Windows Explorer



The project directory holds a total of five subfolders and 16 files at this point. The major file for any CodeWarrior project is its `<project_name>.mcp` file. This is the file to reopen your project.

Return back to the CodeWarrior project window. Double-click on the `main.asm` file in the Sources group. The editor in CodeWarrior opens the `main.asm` file (Figure 1.11 on page 30).

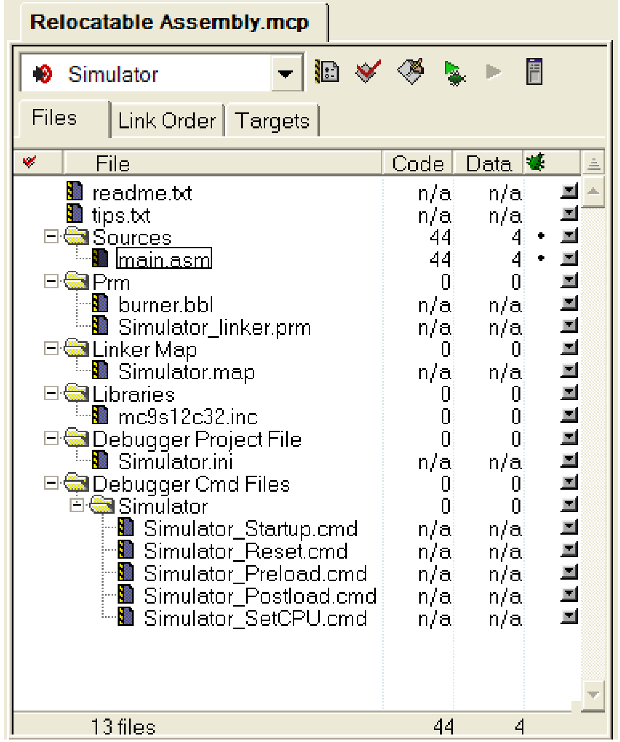
Figure 1.11 main.asm file in the project window

```
main.asm
Path: E:\Projects\Relocatable A...\main.asm

;**************************************************************
;* This stationery serves as the framework for a              *
;* user application. For a more comprehensive program that    *
;* demonstrates the more advanced functionality of this       *
;* processor, please see the demonstration applications       *
;* located in the examples subdirectory of CodeWarrior for    *
;* the HC12 Program directory.
;**************************************************************

; export symbols
            XDEF Entry, main
            ; We use 'Entry' as an export 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

; include derivative specific macros
            INCLUDE 'mc9s12c32.inc'


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

Line 7    Col 31
```

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

As a precaution, you can determine if the project is configured correctly and the source code is free of syntactical errors. It is not necessary that you do so, but you should make (build) the default project that CodeWarrior just created. Either press the *Make* button from the toolbar, or from the Project menu, select (*Project > Make*). All of the red check marks will disappear after a successful building of the project (Figure 1.11).

Figure 1.12 Project window after a successful build



Notice that in the Code and Data columns in the project window show that the code size is 44 bytes and the data size is 4 bytes after assembling the `main.asm` file. If you checked the project directory after the first successful build (make) of the project with the Windows Explorer, you would see that another subfolder and five additional files were created. The new subfolder - *ObjectCode* - holds an object file for every assembly or C/C++ source code file. In this case, the `main.asm.o` file was generated.

Analysis of groups and files in the project window

There are six default groups for this holding this project's files. It really does not matter in which group a file resides as long as that file is somewhere in the project window. A file does not even have to be in any group. The groups do not correspond to any physical folder in the project directory. They are simply present in the project window for

conveniently grouping files anyway you choose. You can add, rename, or delete files or groups, or you can move files or groups anywhere in the project window.

CodeWarrior groups

These groups and their usual functions are:

- Sources

  This group contains the assembly source code files.

- Prm

  This group holds the burner file and the Linker prm file.

- Linker Map

  This group has the Linker Map file.

- Library

  This group holds an include file. This project has an include file for the particular CPU derivative. In this case, the `MC9S12C32.inc` file is for the MC9S12C32 derivative.

- Debugger Project File

  This group holds the `project.ini` file for configuring the debugger.

- Debugger Cmd Files

  A group with five debugger command files are located here.

NOTE The default configuration of the project by the Wizard does not generate an assembler output listing file for any `*.asm` file. However, you can afterwards select the *Generate a listing file* in the assembler options for the Assembler to generate a format-configurable listing file for the assembly source code and include files. Assembler listing files (with `*.lst` file extensions) are usually located in the *bin* subfolder in the project directory when `*.asm` files are assembled with this option set.

TIP To set up your project for generating assembler output listing files, select: *File* > *<Target Name> Settings...* > *Target* > *Assembler for HC12* > *Options* > *Output*. Check *Generate a listing file*. If you want to format the listing files from the default format, check *Configure listing file* and make the desired formatting options. You can also add these listing files to the project window for easier viewing instead of having to continually hunt for them.

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 CodeWarrior. You can begin a new file by pressing the *New Text File* icon on the Toolbar to open a new file, write your assembly-source code, and later save it with a `*.asm` file extension using the *Save* icon on the Toolbar to name and store it wherever you want it placed - usually in the *Sources* folder.

After the assembly-code file is written, it is added to the project using the *Project* menu. If the source file is still open in the project window, select the *Sources* group icon in the project window, single-click on the file that you are writing, and then select *Project > Add <filename> to Project*. The newly created file is then added to the *Sources* group in the project. If you do not first select the destination group's icon (for example, *Sources*) in the project window, the file will most likely be added to the bottom of the files and groups in the project window, which is OK. You can drag and drop the icon for any file wherever and whenever you want in the project window.

Analyzing the project files

We will analyze the default `main.asm` file that was generated when the project was created with the New Project Wizard. Listing 1.2 on page 33 is the default `main.asm` file that is located in the `Sources` folder created by the New Project Wizard.

Listing 1.2 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 CodeWarrior for    *
;* the HC12 Program directory.                                *
;**************************************************************

; export symbols
            XDEF Entry, main
            ; We use 'Entry' as an export 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

; include derivative specific macros
            INCLUDE 'mc9s12c32.inc'


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


; code section
MyCode:     SECTION
main:
Entry:
            LDS  #__SEG_END_SSTACK    ; initialize the stack pointer
            CLI                       ; enable interrupts

EndlessLoop:
            LDX   #1                  ; X contains counter
CounterLoop:
            STX   Counter             ; update global.
            BSR   CalcFibo
            STD   FiboRes             ; store result
            LDX   Counter
            INX
            CPX   #24                 ; Larger values cause overflow.
            BNE   CounterLoop
            BRA   EndlessLoop         ; restart


; Function to calculate Fibonacci numbers. Argument is in X.
CalcFibo:
            LDY   #$00                ; second last
            LDD   #$01                ; last
            ; loop once more (if X was 1, were done already)
            DBEQ  X,FiboDone
FiboLoop:    ; overwrite second last with new value
```

```
            LEAY  D,Y
            ; exchange them -> order is correct again
            EXG   D,Y
            DBNE  X,FiboLoop
FiboDone:
            RTS                        ; Result is in D.
```

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

- Make sure that symbols outside of 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 “XDEF Entry, main” assembly directive 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 when using the Simulator/Debugger.
- Make sure to initialize the stack pointer when using 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 using the New Project Wizard with CodeWarrior initializes the stack pointer automatically with a symbol defined by the Linker for the end of the stack “__SEG_END_SSTACK”.

NOTE An Absolute Assembly project does not require a Linker PRM file as the memory allocation is configured in the projects’s *.asm file instead.

Assembling your source files

Once an assembly source file is available, you can assemble it. You can either utilize CodeWarrior to assemble the *.asm files or alternatively you can use the standalone assembler that is located among the other Build Tools in the prog subfolder of the <CodeWarrior installation> folder.

Assembling with CodeWarrior

CodeWarrior simplifies the assembly of your assembly source code. You can assemble the source code files into its output object (*.o) files (without linking them) by:

- selecting one or more `*.asm` files in the project window and then select *Compile* from the *Project* menu (*Project* > *Compile*). Only `*.asm` files that were preselected will generate updated `*.o` object files.
- select *Project* > *Bring Up To Date*. It is not necessary to preselect any assembly source files when using this command.

The object files are generated and placed into the `ObjectCode` subfolder in the project directory.

NOTE The target name can be changed to whatever you choose in the *Target Settings* (preference) panels. Select *Edit* > *<target_name> Settings...* > *Target* > *Target Settings* and enter the revised target name into the *Target Name:* text box. The default *<target_name>* is *Simulator*.

Or, you can assemble all the `*.asm` files and link the resulting object files (and any appropriate library files) to generate the executable `<target_name>.abs` file by invoking either *Make* or *Debug* from the *Project* menu (*Project* > *Make* or *Project* > *Debug*). This results in the generation of the `<target_name>.abs` file in the `bin` subfolder of the project directory.

Two other files generated by CodeWarrior after linking (*Make*) or *Debug* are:

- `<target_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.

- `<target_name>.abs.s19`

  This is an S-Record File that can be used for programming a ROM memory.

TIP The remaining file in the default `bin` subfolder is the `main.dbg` file that was generated back when the `main.asm` file was successfully assembled. This debugging file was generated because a bullet was present in the debugging column in the project window.
You can enter (or deselect by subsequently toggling) a debugging bullet by clicking at the intersection of the `main.asm` file (or whatever other source code file selected for debugging) and the debugging column in the project window.
Whenever the Debugger or Simulator does not show a desired file in its `Source` window, check first to see if the debugging bullet is present or not in the project window. The bullet must be present for debugging purposes.

TIP The New Project Wizard does not generate default assembler-output listing files. If you want such listing files generated, you have to select this option:
Edit > *<target_name> Settings* > *Target* > *Assembler for HC12* > *Options*. Select the *Output* tab in the *HC12 Assembler Option Settings* dialog box. Check *Generate a listing file* and *Do not print included files in list file*. (You can uncheck

Do not print included files in list file if you choose, but be advised that the include files for CPU derivatives are usually quite lengthy.) Now a `*.lst` file will be generated or updated in the `bin` subfolder of the project directory whenever a `*.asm` file is assembled.

TIP You can also 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.

Assembling with the Assembler

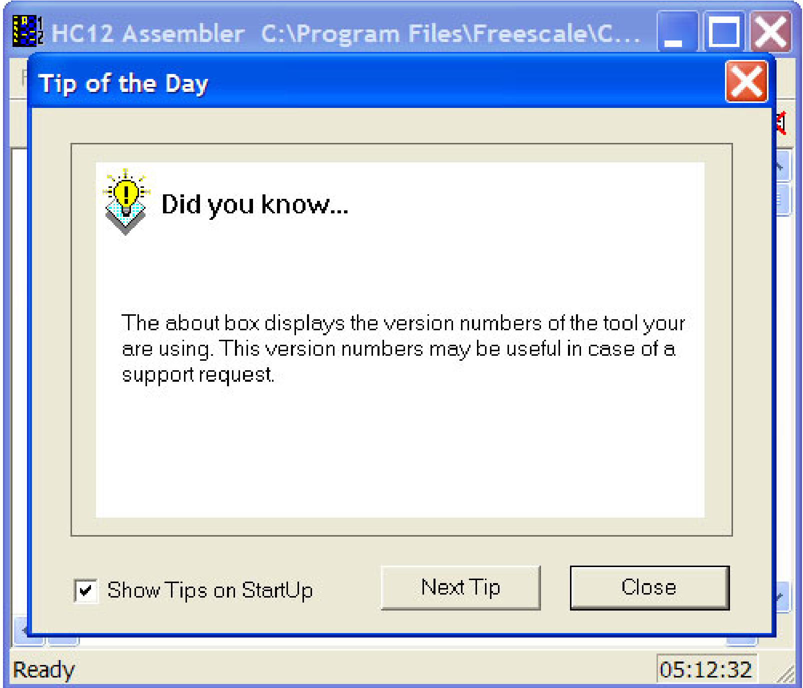
It is also possible to use the HC(S)12 Assembler as a standalone assembler. (If you already have an assembled source file and prefer not to use the Assembler but do want to use the Linker, you can skip this section and proceed to "Linking the application" on page 50.)

This tutorial does not create another project with the Build Tools, but instead makes use of a project already created by the CodeWarrior New Project Wizard. CodeWarrior can create, configure, and manage a project much easier and quicker than using the Build Tools. However, the Build Tools could also create and configure an entire project from scratch.

A Build Tool such as the Assembler uses a project directory file for configuring and locating its generated files. The folder that is set up for this purpose is referred to by a Build Tool as the "current directory."

Start the Assembler. You can do this by opening the `ahc12.exe` file in the `prog` folder in the HC12 CodeWarrior installation. The Assembler opens (Figure 1.13 on page 38).

Figure 1.13 HC12 Assembler opens...



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

Configuring the Assembler

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

- Global

  This data is common to all Build Tools and projects. There may be common data for each Build Tool (Assembler, Compiler, Linker, ...) such as listing the most recent projects, etc. All tools may store some global data in the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (e.g. `C:\WINDOWS`). See Listing 1.3 on page 39.

Listing 1.3 Typical locations for a global configuration file

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

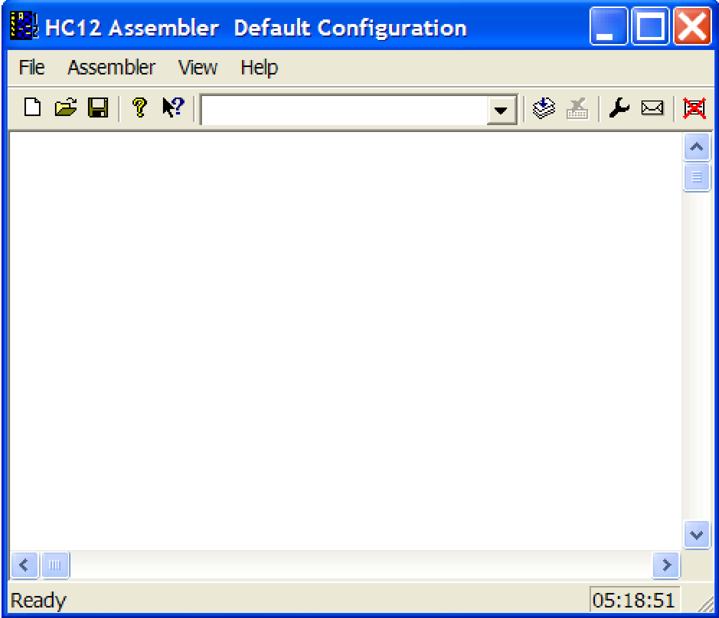
For information about entries for the global configuration file, see Global Configuration File Entries on page 399 in the Appendices.

- Local

  This file could be used by any Build Tool for a particular project. For information about entries for the local configuration file, see Local Configuration File Entries on page 409 in the Appendices.

After opening the assembler, you would load the configuration file for your project if it already had one. In this case, you will create a new configuration file and save it so that whenever the project is reopened, its previously saved configuration state will be used. From the *File* menu, select *New / Default Configuration*. The *HC12 Assembler Default Configuration* dialog box appears (Figure 1.14 on page 39)

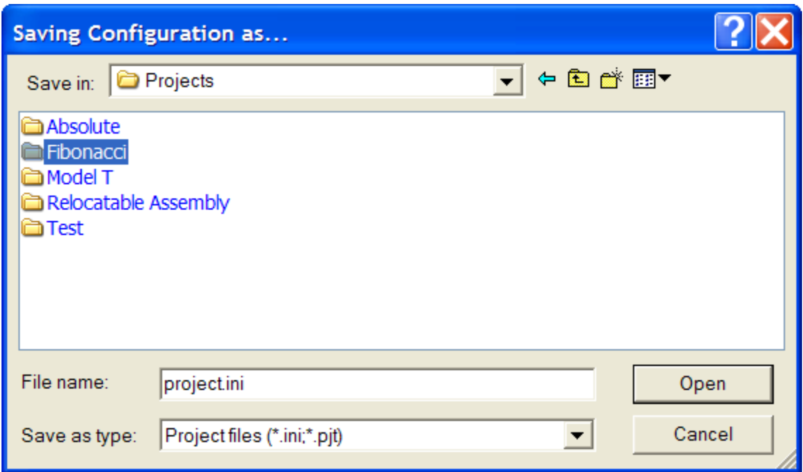
Figure 1.14 HC12 Assembler Default Configuration dialog box



Now let's save this configuration in a newly created folder that will become the project directory. From the *File* menu, select *Save Configuration*. A *Saving Configuration as...*

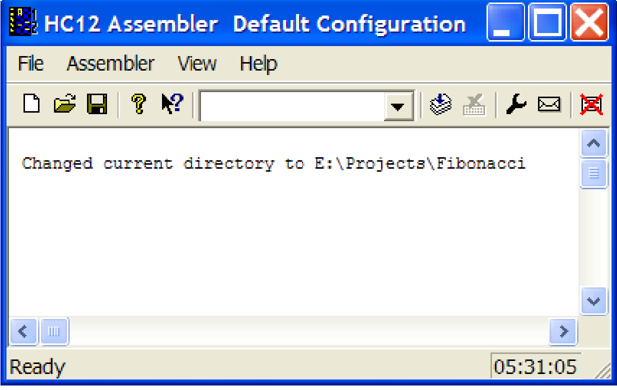
dialog box appears. Navigate to the folder of your choice and create and name a folder and filename for the configuration file (Figure 1.15 on page 40).

Figure 1.15 Loading configuration dialog box



Press *Open*. The current directory for the HC(S)12 Assembler changes to your project directory (Figure 1.16 on page 40).

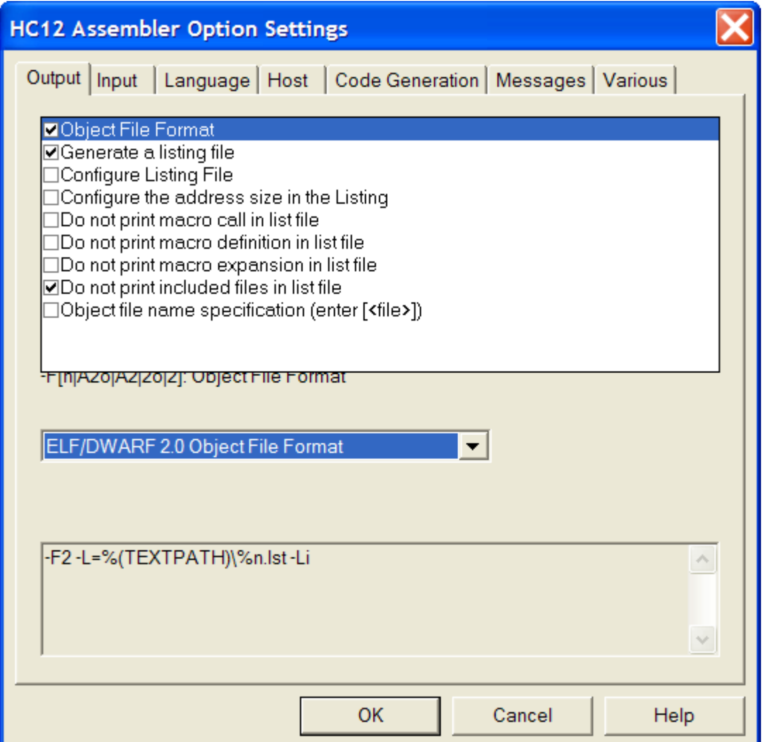
Figure 1.16 Assembler's current directory switches to your project directory...



If you were to examine the project directory with the Windows Explorer at this point, it would only contain the *<project_name>.ini* configuration file that you just created. Any options added to or deleted from your project by any Build Tool would be placed into or deleted from this configuration file in the appropriate section for each Build Tool.

You now set the object-file format that you intend to use (HIWARE or ELF/DWARF). Select the menu entry *Assembler > Options*. The Assembler displays the *HC12 Assembler Option Settings* dialog box (Figure 1.17 on page 41).

Figure 1.17 HC12 Assembler Option Settings dialog box



In the *Output* panel, select the check boxes labeled *Generate a listing file* and *Object File Format*. For the *Object File Format*, select the *ELF/DWARF 2.0 Object File Format* in the pull-down menu. The listing file would be much shorter if the *Do not print included files in list file* check box is checked, so you may want to select that option also. Press *OK* to close the *HC12 Assembler Option Settings* dialog box.

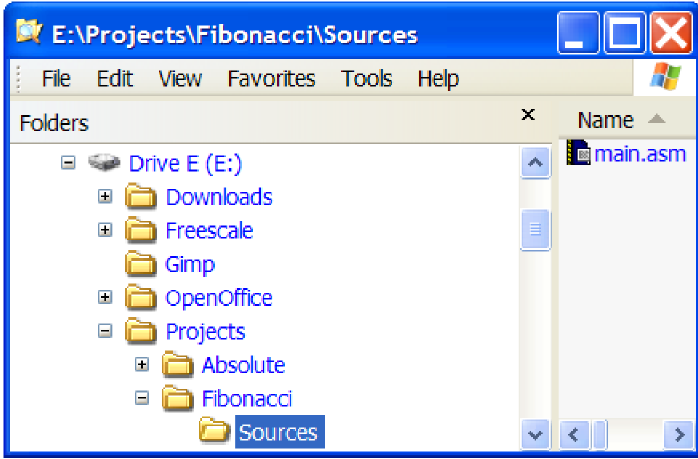
Save the changes to the configuration by:

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

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 `Sources` folder from the previous CodeWarrior project into the project directory (Figure 1.18 on page 42).

Figure 1.18 Project files



Now there are two files in the project:

- the `project.ini` configuration file and
- `main.asm` in the `Sources` folder:

The contents of the `main.asm` file are displayed in Listing 1.4 on page 42.

Listing 1.4 main.asm file

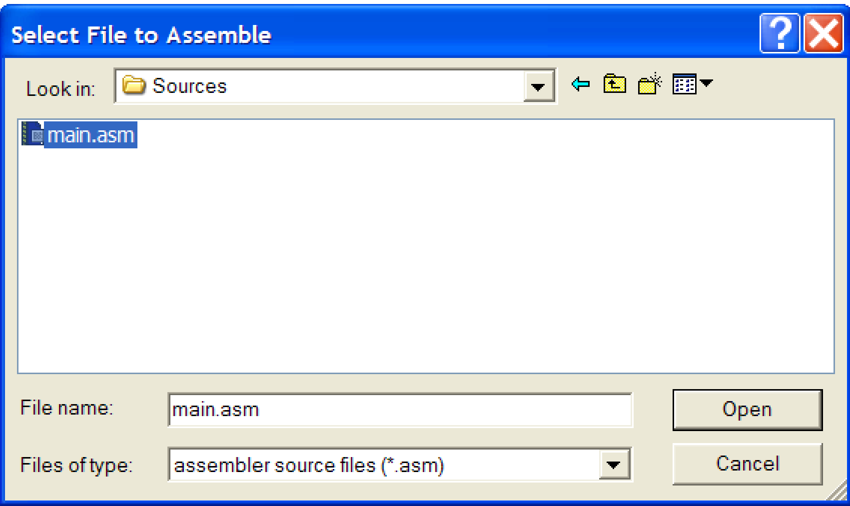
```
;*************************************************************
;* This stationery serves as the framework for a             *
;* user application. For a more comprehensive program that   *
;* demonstrates the more advanced functionality of this      *
;* processor, please see the demonstration applications      *
;* located in the examples subdirectory of CodeWarrior for   *
;* the HC12 Program directory.                               *
;*************************************************************

; export symbols
            XDEF Entry, main
```

```
            ; We use 'Entry' as an export 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

; include derivative specific macros
            INCLUDE 'mc9s12c32.inc'


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


; code section
MyCode:     SECTION
main:
Entry:
            LDS   #__SEG_END_SSTACK   ; initialize the stack pointer
            CLI                       ; enable interrupts

EndlessLoop:
            LDX   #1                  ; X contains counter
CounterLoop:
            STX   Counter             ; update global.
            BSR   CalcFibo
            STD   FiboRes             ; store result
            LDX   Counter
            INX
            CPX   #24                 ; Larger values cause overflow.
            BNE   CounterLoop
            BRA   EndlessLoop         ; restart
```

Assembling the Assembly source-code files

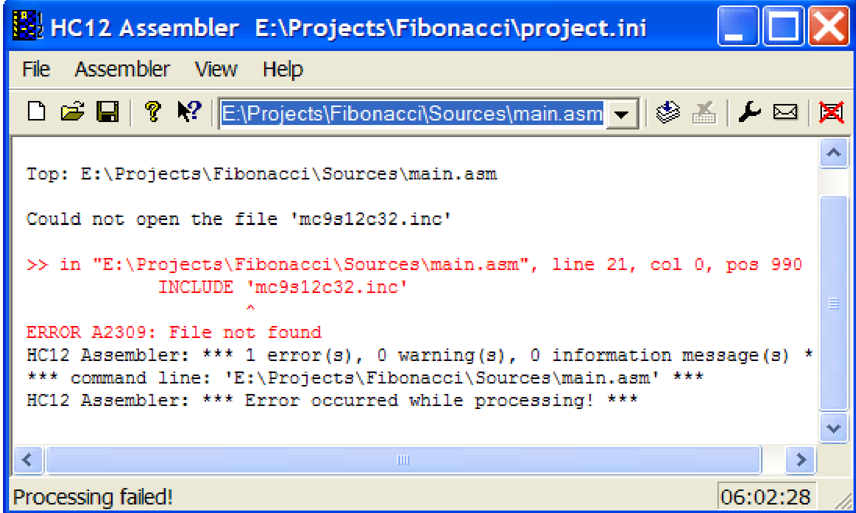
Let’s assemble the `main.asm` file. From the *File* menu, select *Assemble*. The *Select File to Assemble* dialog box appears (Figure 1.19 on page 44).

Figure 1.19 Select File to Assemble dialog box



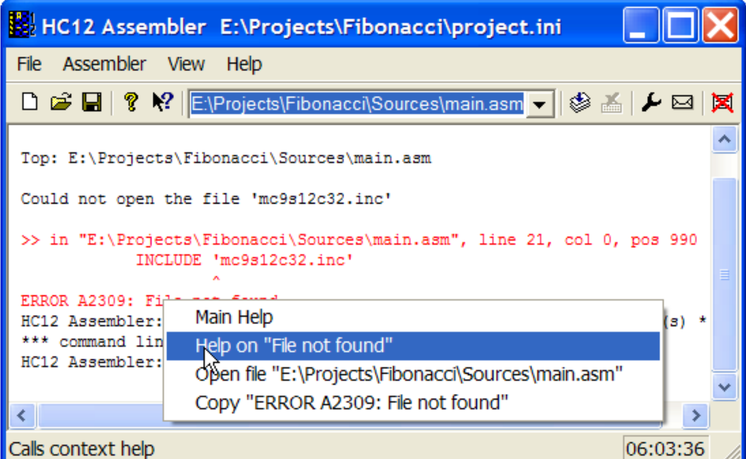
Browse to the `Sources` folder in the project directory and select the `main.asm` file. Press *Open* and the `main.asm` file should start assembling (Figure 1.20 on page 44).

Figure 1.20 Results of assembling the main.asm file...



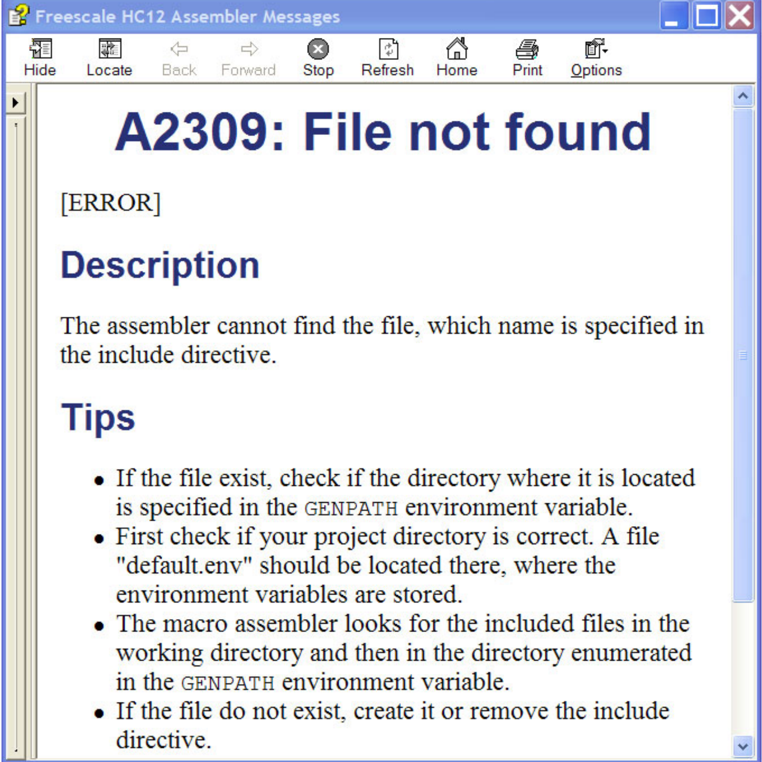
The project window provides positive information about the assembly process or generates error messages if the assembly was unsuccessful. In this case an error message is generated. - the *A2309 File not found* message. If you right-click on the text about the error message, a context menu appears (Figure 1.21 on page 45).

Figure 1.21 Context menu



Select *Help on "file not found"* and Help for the A2309 error message appears (Figure 1.22 on page 46).

Figure 1.22 A2309 error message help

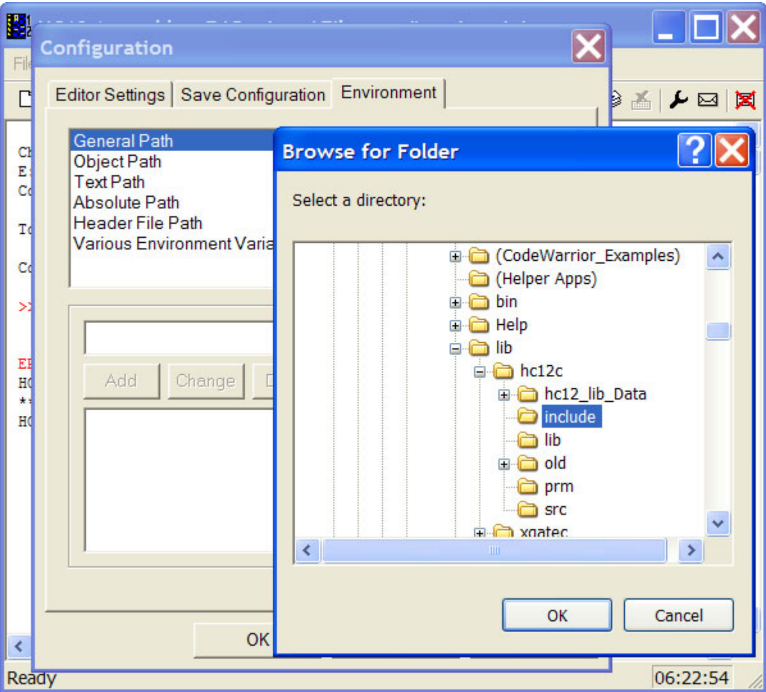


You know that the file exists because it is included in the `Sources` folder that you imported into the project directory. The help message for the A2309 error states that the Assembler looks for this "missing" include file first in the current directory and then in the directory specified by the `GENPATH` environment variable. This implies 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 21: INCLUDE 'mc9s12c32.inc'.

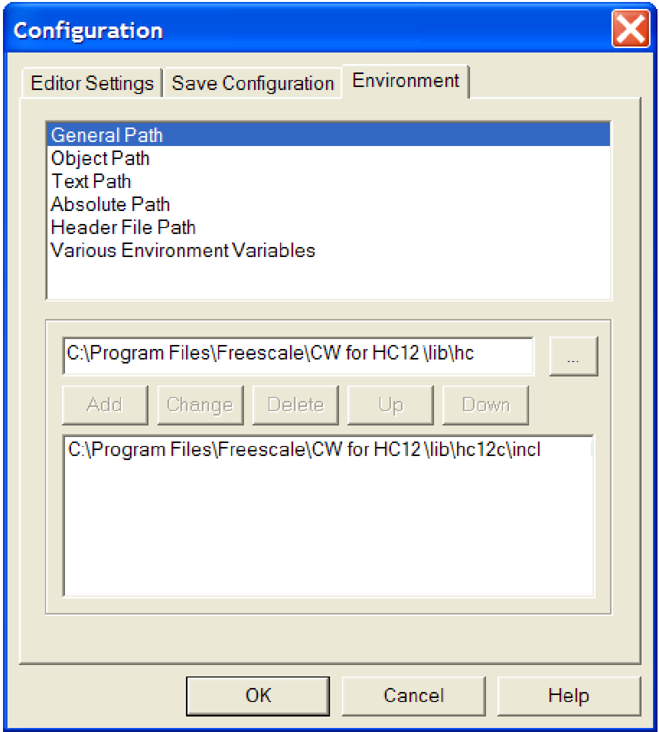
To fix this, select *File > Configuration*. The *Configuration* dialog box appears (Figure 1.23 on page 47).

Figure 1.23 Browsing for the Sources folder



Select the *Environment* tab and then *General Path*. Press the "..." button and navigate in the *Browse for Folder* dialog box for the folder that contains the missing file - the `include` subfolder in the CodeWarrior installation's `lib` folder. Press *OK* to close the *Browse for Folder* dialog box. The *Configuration* dialog box is now again active (Figure 1.24 on page 48).

Figure 1.24 Adding a GENPATH



Press the *Add* button, and the path to the `mc9s12c32.inc` file "*{Compiler}\lib\hc12c\include*" now appears in the lower panel. Press *OK*. An asterisk now appears in the Title bar, so save the change to the configuration by pressing the *Save* button or by selecting *File > Save Configuration*. The asterisk disappears when the file is saved.

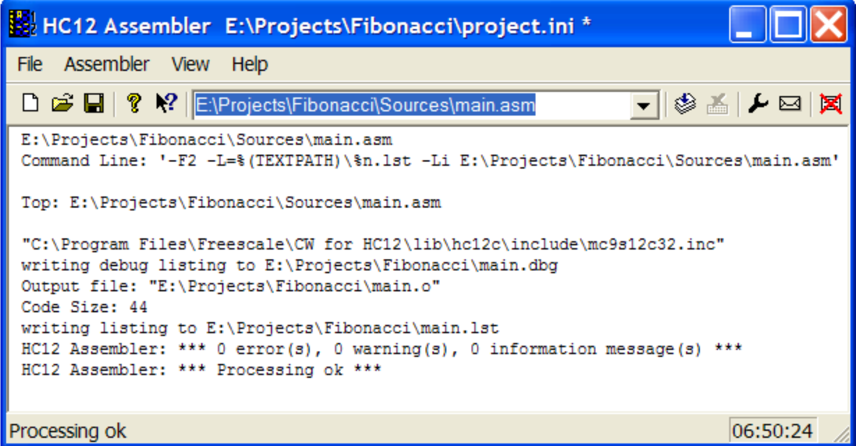
TIP You can clear the messages in the Assembler window at any time by selecting *View > Log > Clear Log*.

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

Select *File > Assemble* and again navigate to the *`main.asm`* file and press *Open*.

After the `GENPATH` is set up for the include file, you can try to assemble the `main.asm` file again (Figure 1.25 on page 49).

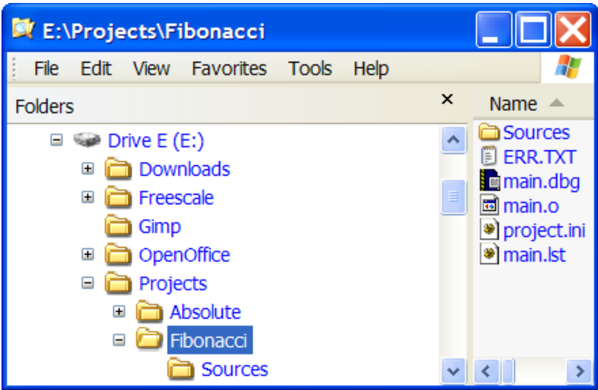
Figure 1.25 Successful assembly - main.o object file created



The Macro Assembler indicates successful assembling and indicated that the Code Size is 44 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 Macro Assembler 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 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`. The assembly output file is similarly named - `main.lst`. The `ERR.TXT` file was generated as a result of the first failed attempt to assemble the main.asm file without the correct path to the *.inc file (Figure 1.26 on page 50).

Figure 1.26 Project directory after a successful assembly



The haphazard running of this project was intentionally designed to fail in order to illustrate what would occur 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.

So in the future, read through the `*.asm` files before assembling and set up whatever paths are required for any include (`*.inc`) files. If there were more than one `*.asm` file in the project, you could select any or all of them, and the selected `*.asm` files would be assembled simultaneously.

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

Linking with CodeWarrior

If you are using CodeWarrior to manage your project, a pre-configured PRM file for a particular derivative is already set up (Listing 1.5 on page 50).

Listing 1.5 Linker PRM file for the MC9S12C32 derivative

```
/* This is a linker parameter file for the MC9S12C32 */
NAMES END /* CodeWarrior will pass all the needed files to the
   linker by command line. But you may add your own files here too. */

SEGMENTS /* All RAM/ROM areas of the device are listed. Used in
```

```
            PLACEMENT below. */
    RAM      = READ_WRITE 0x0800 TO 0x0FFF;
    /* unbanked FLASH ROM */
    ROM_4000 = READ_ONLY  0x4000 TO 0x7FFF;
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
    /* banked FLASH ROM */
/*   PAGE_3E = READ_ONLY  0x3E8000 TO 0x3EBFFF; not used: equivalent
  to ROM_4000 */
/*   PAGE_3F = READ_ONLY  0x3F8000 TO 0x3FBFFF; not used: equivalent
  to ROM_C000 */
// OSVECTORS = READ_ONLY  0xFF8A TO 0xFFFF; /* OSEK interrupt vectors
                                            (use your vector.o) */
END

PLACEMENT /* 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                     /* copy down information: how to
                                initialize variables */
                             /* In case you want to use ROM_4000
                                here as well, make sure
                                that all files (incl. library
                                files) are compiled with the
                                option: -OnB=b */
                                 INTO  ROM_C000/*, ROM_4000*/;

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

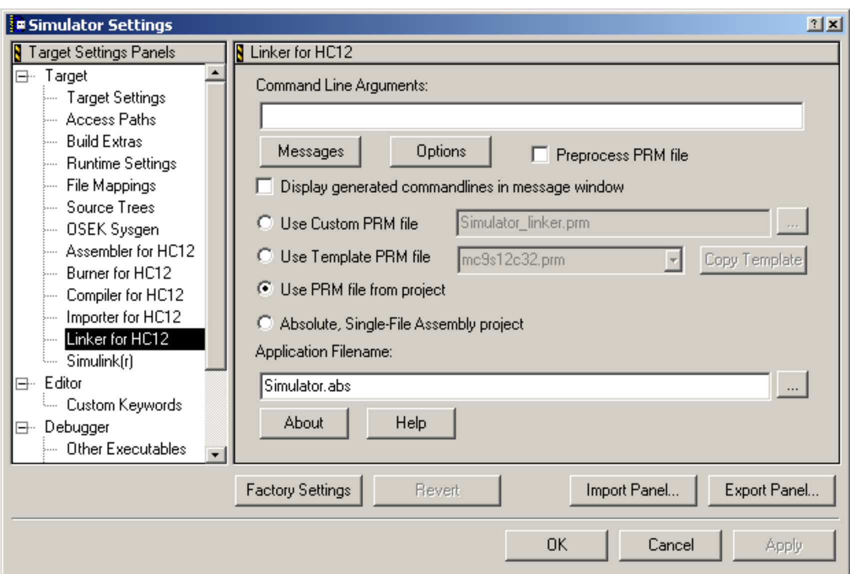
```
END

STACKSIZE 0x80

//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: This is also
                       the initialization entry point */
```

NOTE A number of entries in the PRM file in Listing 1.5 on page 50 are “commented-out” by the CodeWarrior IDE because they would not be utilized in this simple relocatable assembly project.

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`. You may use a different PRM file in place of the default PRM file that was generated by the New Project Wizard.

The *Linker for HC12* preference panel controls which PRM file is used for your CodeWarrior project. The default PRM file for a CodeWarrior project is the PRM file in the project window. Let’s see what other options exist for the PRM file. From the *Edit* menu, select *<target_name> Settings... > Target > Linker for HC12*. The *Linker for HC12* preference panel appears (Figure 1.27 on page 53).

Figure 1.27 Linker for HC12 preference panel



There are three radio buttons for selecting the PRM file and another for selecting an absolute, single-file assembly project:

- *Use Custom PRM file* (exists for backward compatibility)
- *Use Template PRM file* (exists for backward compatibility)
- *Use PRM file from project* - the default, or
- *Absolute, Single-File Assembly project*.

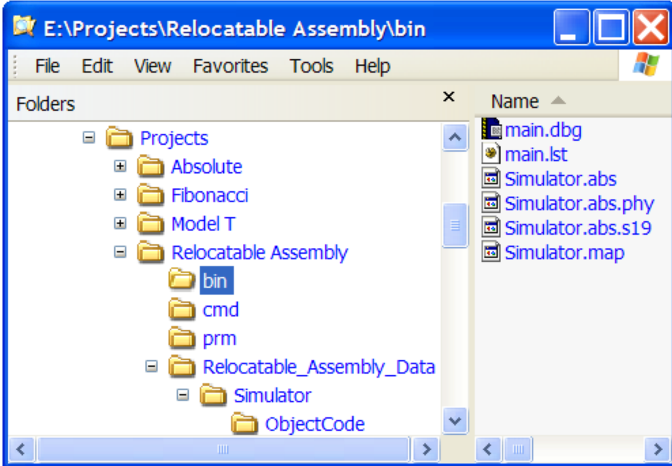
In case you want to change the filename of the application, you can determine the filename and its path with the *Application Filename:* text box. See the Smart Linker section of the “Build Tools” manual for details.

The ‘`STACKSIZE`’ entry is used to set the stack size. The size of the stack for this project is 80 bytes. The `Entry` symbol is used for both the entry point of the application and for the initialization entry point.

Linking the object-code files

You can run this relocatable assembly project from the Project menu: Select *Project > Make* or *Project > Debug*. The Linker generates a `*.abs` file and a `*.abs.s19` standard S-Record File in the `bin` subfolder of the project directory. You can use the S-Record File for programming a ROM memory (Figure 1.28 on page 54).

Figure 1.28 Project directory in Windows Explorer after linking



The *Full Chip Simulation* option in CodeWarrior was selected when the project was created, so if *Project > Debug* is selected, the debugger opens and you can follow each assembly-code instruction during the execution of the program with the Simulator. You can single-step the simulator through the program's assembly-source instructions from the *Run* menu in the Simulator (*Run > Assembly Step* or *Ctrl+F11*).

Linking with the Linker

If you are using the standalone Linker, you will use a PRM file for the Linker to allocate memory.

- Start your editor and create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as `<target_name>.prm`.
- Store the PRM file in a convenient location. A good spot would be directly into the project directory.
- In the `<target_name>.prm` file, add the name of the executable (`*.abs`) file, say `<target_name>.abs`. In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's `Fibonacci.prm` file — a PRM file for an MC9S12C32 from another CodeWarrior project was adapted — is shown in Listing 1.6 on page 54.

Listing 1.6 Layout of a PRM file for the Linker - Fibonacci.prm

```
/* This is a linker parameter file for the MC9S12C32 */
LINK Fibonacci.abs
NAMES main.o /* filenames of the object files to be linked */
```

```
END

SEGMENTS /* All RAM/ROM areas of the device are listed. Used in
            PLACEMENT below. */
    RAM      = READ_WRITE 0x0800 TO 0x0FFF;
    /* unbanked FLASH ROM */
    ROM_4000 = READ_ONLY  0x4000 TO 0x7FFF;
    ROM_C000 = READ_ONLY  0xC000 TO 0xFEFF;
END

PLACEMENT /* All predefined and user segments are placed into the
             SEGMENTS defined above. */
    ROM_VAR,                 /* constant variables */
    NON_BANKED,              /* runtime routines which must not be
                                banked */
    DEFAULT_ROM                 INTO  ROM_C000/*, ROM_4000*/;

    .stack,                  /* allocate stack first to avoid
                                overwriting variables on overflow */
    DEFAULT_RAM                 INTO  RAM;
END

STACKSIZE 0x100

VECTOR 0 Entry      /* reset vector: This is the default entry point
                       for an Assembly application. */
INIT Entry          /* for assembly applications: This is also
                       the initialization entry point */
```

NOTE If you are adapting a PRM file from a CodeWarrior project, most of what you need to do is add object filenames that are to be linked in the `LINK` portion and `NAMES` portion.

NOTE The default size for the stack using the CodeWarrior New Project Wizard for the MC9S12C32 is 256 bytes: (STACKSIZE 0x100). This command and `__SEG_END_SSTACK` in the assembly code file determine the size and placement of the stack in RAM:

```
MyCode:     SECTION
main:
Entry:
    LDS #__SEG_END_SSTACK ; initialize the stack ptr
```

The commands in the linker parameter file are described in the Linker portion of the Build Tools manual.

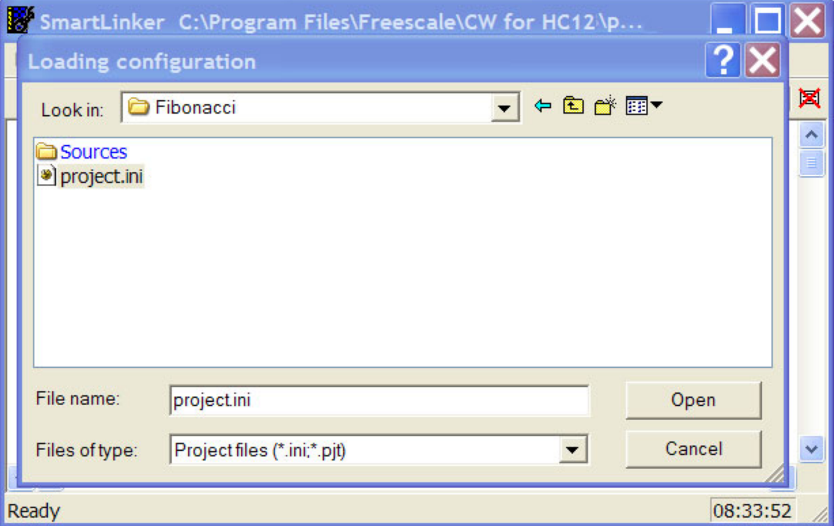
- Start the Linker.

  The Smart Linker tool is located in the `prog` folder in the CodeWarrior installation: `proj\linker.exe`

- Press *Close* to close the *Tip of the Day* dialog box.
- Load the project's configuration file. Use the same `<project>.ini` that the Assembler used for its configuration - the `project.ini` file in the project directory:

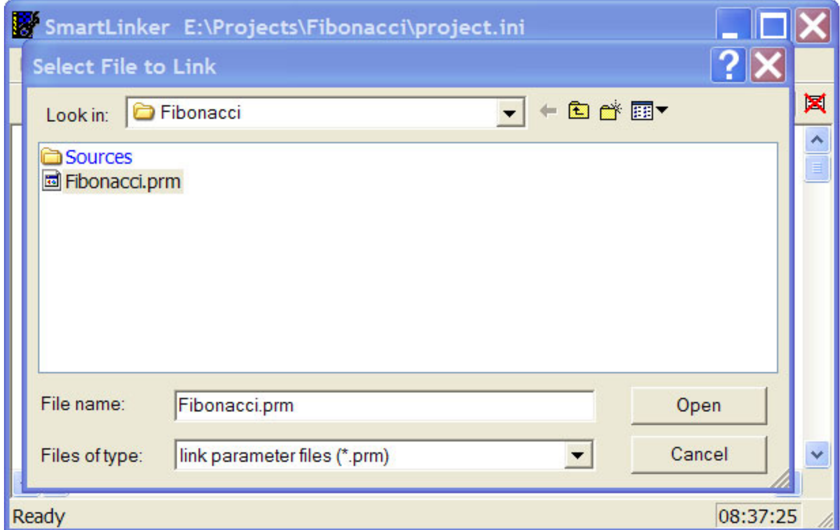
  File > Load Configuration and navigate to the project's configuration file (Figure 1.29 on page 56).

Figure 1.29 HC(S)12 Linker



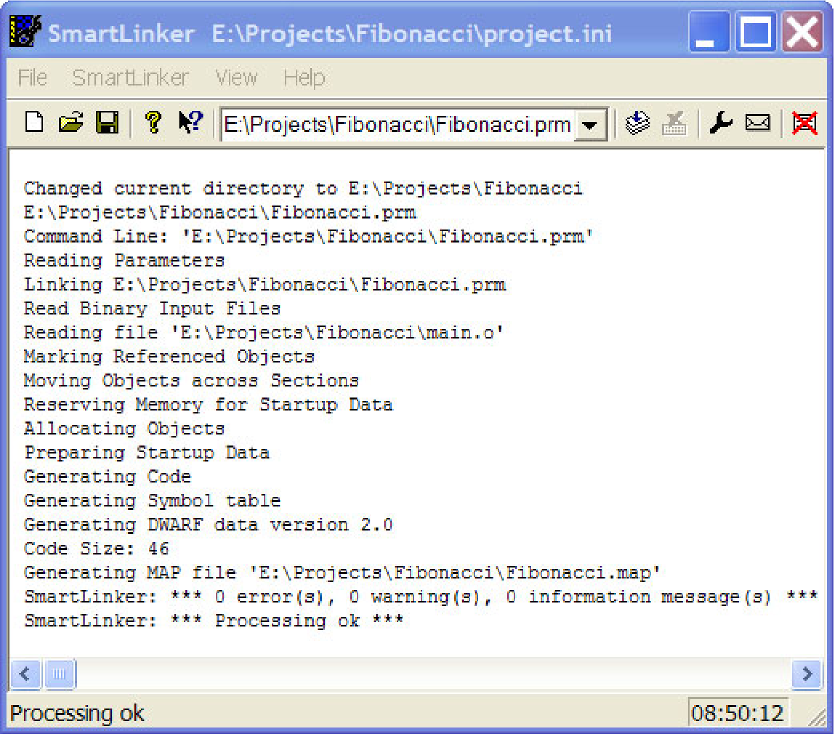
- Press *Open* to load the configuration file. The project directory is now the current directory for the Linker. You can press the *Save* button to save the configuration if you choose. From the *File* menu in the Smart Linker, select *Link*: (*File > Link* (Figure 1.30 on page 57).

Figure 1.30 Select File to Link dialog box



- Browse to locate the PRM file for your project. Select the PRM file. Press *Open*. The Smart Linker links the object-code files in the `NAMES` section to produce the executable `*.abs` file as specified in the `LINK` portion of the Linker PRM file (Figure 1.31 on page 58).

Figure 1.31 Linker main window after linking



The messages in the linker's project window indicate:

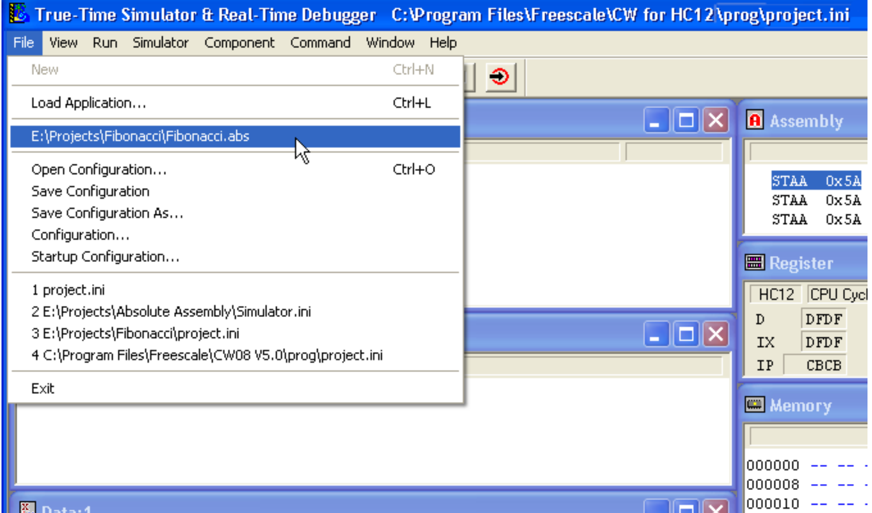
- The current directory for the Linker is the project directory, `E:\Projects\Fibonacci`
- The `Fibonacci.prm` file was used to name the executable file, which object files were linked, and how the RAM and ROM memory areas are to be allocated for the relocatable sections. The Reset and application entry points were also specified in this file.
- There was one object-code file, `main.o`.
- The output format was DWARF 2.0.
- The Code Size was 46 bytes.
- A Linker Map file was generated - `Fibonacci.map`.
- No errors or warnings occurred and no information messages were issued.

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

- Start the Simulator.
- Load the absolute executable file:
  - *File > Load Application...* and browse to the appropriate `*.abs` file, or
  - Select the given path to the executable file, if it is appropriate as in this case (Figure 1.32 on page 59):

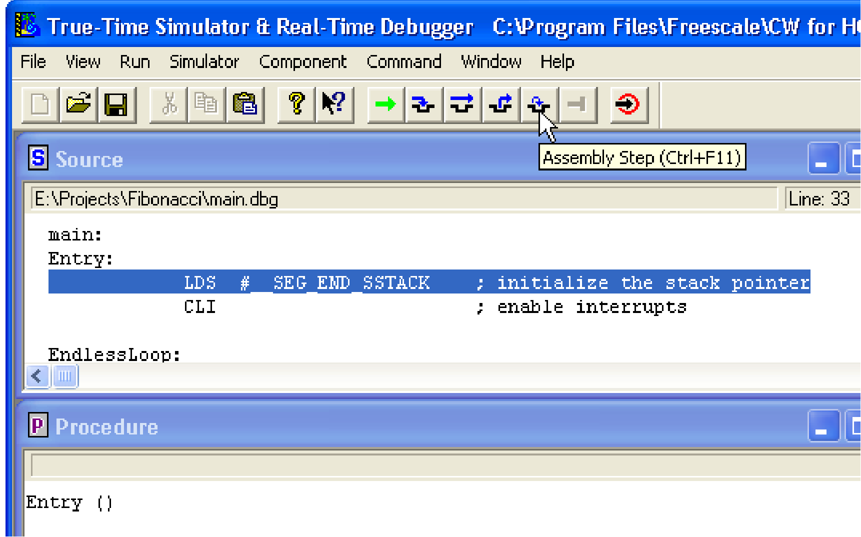
    `E:\Projects\Fibonacci\Fibonacci.abs`

Figure 1.32 HC(S)12 Simulator: Select the executable file



- Assembly-Step (Figure 1.33 on page 60) through the program source code (or do something else...).

Figure 1.33 Assembly Stepping...



Directly generating an ABS file

You can also use CodeWarrior or the standalone assembler to generate an `ABS` file directly from your assembly source file. The Assembler may also be configured to generate an S-Record File at the same time. You can use the S-Record File for programming ROM memory.

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

Using CodeWarrior to generate an ABS file

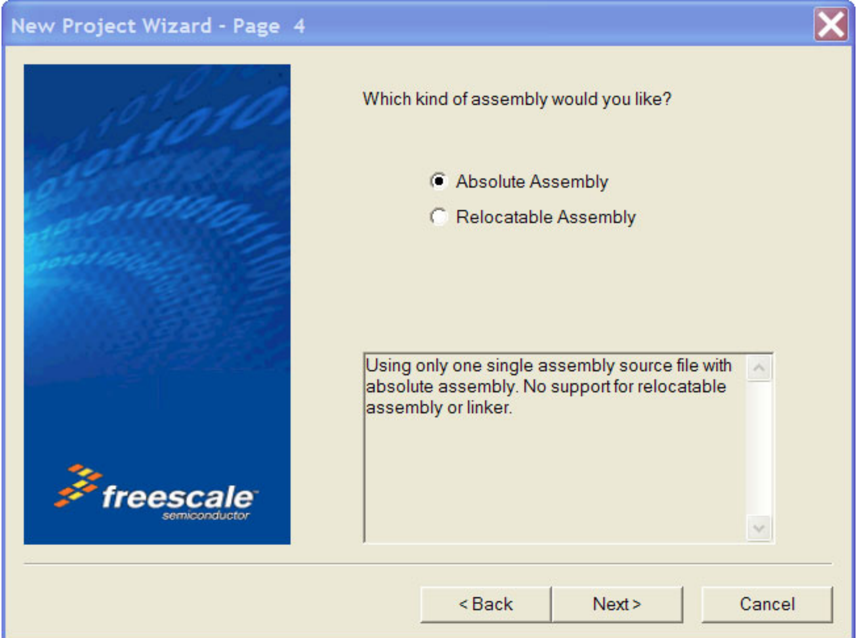
You can use the Wizard to produce an absolute assembly project. To do so, you follow the same steps in creating a relocatable-assembly project given earlier. There are some differences:

- No PRM file is required, so no PRM file will be included in the `Prm` group in the project window.
- The memory area allocations are determined directly in the single `*.asm` assembly source-code file.

Start the CodeWarrior New Project Wizard and create an assembler project in the usual manner. Confer "The New Project Wizard" on page 19 if you need assistance in creating a

CodeWarrior project. However in the *New Project Wizard - Page 4* dialog box, *Absolute Assembly* is selected. That is the only difference between relocatable and absolute assembly using the New Project Wizard (Figure 1.34 on page 61).

Figure 1.34 New Project Wizard - Page 4 dialog box



The single absolute-assembly main.asm file

Only one `*.asm` assembly source-code file can be used in an absolute-assembly project. The `main.asm` source code file differs slightly from a file used in relocatable assembly (Listing 1.7 on page 62).

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

Listing 1.7 main.asm file - absolute assembly

```
;*****************************************************************
;* This stationery serves as the framework for a                *
;* user application (single file, absolute assembly application) *
;* For a more comprehensive program that                         *
;* demonstrates the more advanced functionality of this          *
;* processor, please see the demonstration applications          *
;* located in the examples subdirectory of the                   *
;* CodeWarrior for the HC12 Program directory                    *
;*****************************************************************

; export symbols
            XDEF Entry          ; export 'Entry' symbol
            ABSENTRY Entry      ; for absolute assembly: Mark this
                                ; as the application entry point.

; include derivative specific macros
            INCLUDE 'mc9s12c32.inc'

ROMStart    EQU   $4000 ; absolute address to place my code/constant

; variable/data section
            ORG RAMStart        ; Insert here your data definition.
Counter     DS.W 1
FiboRes     DS.W 1


; code section
            ORG   ROMStart
Entry:
            LDS   #RAMEnd+1     ; initialize the stack pointer
            CLI                 ; enable interrupts
mainLoop:
            LDX   #1            ; X contains counter
counterLoop:
            STX   Counter       ; update global.
            BSR   CalcFibo
            STD   FiboRes       ; store result
            LDX   Counter
            INX
            CPX   #24           ; larger values cause overflow.
            BNE   counterLoop
            BRA   mainLoop      ; restart.

CalcFibo: ; Function to calculate Fibonacci numbers. Argument is in X
            LDY   #$00          ; second last
            LDD   #$01          ; last
```

```
            DBEQ  X,FiboDone      ; loop once more (if X was 1, were
FiboLoop:
            LEAY  D,Y             ; overwrite second last with new va
            EXG   D,Y             ; exchange them -> order is correct
            DBNE  X,FiboLoop
FiboDone:
            RTS                   ; result in D


;**************************************************************
;*                 Interrupt Vectors                          *
;**************************************************************
            ORG   $FFFE
            DC.W  Entry           ; Reset Vector
```

Pay special attention to the following points:

- The Reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application's entry point address is created at the Reset vector address. To set the entry point of the application at address `$FFFE` on the `Entry` symbol, the following code is used (Listing 1.9 on page 63):

Listing 1.8 Using ORG to set the Reset vector

```
            ORG   $FFFE
            DC.W  Entry           ; Reset Vector
```

- The `ABSENTRY` directive is used to write the address of the application entry point in the generated absolute file. To set the entry point of the application on the `Entry` label in the absolute file, the following code is used (Listing 1.9 on page 63).

Listing 1.9 Using ABSENTRY to enter the entry-point address

```
          ABSENTRY Entry
```

Assembling main.asm

From the *Project* menu, select *Bring Up To Date* or select the `main.asm file` in the project window and select *Compile*. If the project's preferences are set to create an assembler output listing file, this will generate a listing file as shown in Listing 1.10 on page 64.

Listing 1.10 Assembler output listing file of main.asm

```
Freescale HC12-Assembler
(c) Copyright Freescale 1987-2005

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ;***********************************
    2    2                      ;* This stationery serves as the fram
    3    3                      ;* user application (single file, abs
    4    4                      ;* For a more comprehensive program t
    5    5                      ;* demonstrates the more advanced fun
    6    6                      ;* processor, please see the demonstr
    7    7                      ;* located in the examples subdirecto
    8    8                      ;* Freescale CodeWarrior for the HC1
    9    9                      ;***********************************
   10   10
   11   11                      ; export symbols
   12   12                                  XDEF Entry           ; e
   13   13                                  ABSENTRY Entry       ; f
   14   14                                                       ; a
   15   15
   16   16                      ; include derivative specific macros
   17   17                                  INCLUDE 'mc9s12c32.inc'
 5396   18
 5397   19          0000 4000   ROMStart    EQU   $4000  ; absolute a
 5398   20
 5399   21                      ; variable/data section
 5400   22                                  ORG RAMStart         ; I
 5401   23  a000800             Counter     DS.W 1
 5402   24  a000802             FiboRes     DS.W 1
 5403   25
 5404   26
 5405   27                      ; code section
 5406   28                                  ORG   ROMStart
 5407   29                      Entry:
 5408   30  a004000 CF10 00                 LDS   #RAMEnd+1     ; i
 5409   31  a004003 10EF                    CLI                  ;
 5410   32                      mainLoop:
 5411   33  a004005 CE00 01                 LDX   #1             ; X
 5412   34                      counterLoop:
 5413   35  a004008 7E08 00                 STX   Counter        ; u
 5414   36  a00400B 070E                    BSR   CalcFibo
 5415   37  a00400D 7C08 02                 STD   FiboRes        ; s
 5416   38  a004010 FE08 00                 LDX   Counter
 5417   39  a004013 08                      INX
 5418   40  a004014 8E00 18                 CPX   #24            ; L
```

```
5419   41  a004017 26EF                  BNE    counterLoop
5420   42  a004019 20EA                  BRA    mainLoop        ; r
5421   43
5422   44                    CalcFibo:  ; Function to calculate Fi
5423   45  a00401B CD00 00               LDY    #$00            ; s
5424   46  a00401E CC00 01               LDD    #$01            ; l
5425   47  a004021 0405 07               DBEQ   X,FiboDone      ; l
5426   48                    FiboLoop:
5427   49  a004024 19EE                  LEAY   D,Y             ; o
5428   50  a004026 B7C6                  EXG    D,Y             ; e
5429   51  a004028 0435 F9               DBNE   X,FiboLoop
5430   52                    FiboDone:
5431   53  a00402B 3D                    RTS                    ; r
5432   54
5433   55
5434   56                    ;************************************
5435   57                    ;*                 I
5436   58                    ;************************************
5437   59                                ORG    $FFFE
5438   60  a00FFFE 4000                  DC.W   Entry           ; R
```

However, using the *Bring Up To Date* or *Compile* commands will not produce an executable (`*.abs`) output file. From the *Project* menu, select either *Make* or *Debug* (*Project > Make* or *Project > Debug*) to generate the `*.abs` executable and `*.abs.s19` files in the `bin` subfolder. Be advised that it is not necessary to use the *Compile* or *Bring Up To Date* commands used earlier to produce an assembler output listing file because using either the *Make* or *Debug* command also performs that functionality.

If you want to analyze the logic of the Fibonacci program, you can use the Simulator/ Debugger and assemble-step it through the program. If you select *Project > Debug*, the Simulator opens and you can follow the execution of the program while assemble-stepping the Simulator either from the *Run* menu in the Simulator (*Run > Assembly Step* or *Ctrl + F11*).

Using the Assembler for absolute assembly

Create a new configuration *project.ini* file and directory for the absolute assembly project using the standalone Assembler Build Tool. This section does not go into the detail that was done for the relocatable assembly section. Use an absolute assembly source file of the type listed in Listing 1.11 on page 65.

Listing 1.11 Main.asm file for absolute assembly

```
;*****************************************************************
;* This stationery serves as the framework for a                 *
;* user application (single file, absolute assembly application) *
;*****************************************************************
```

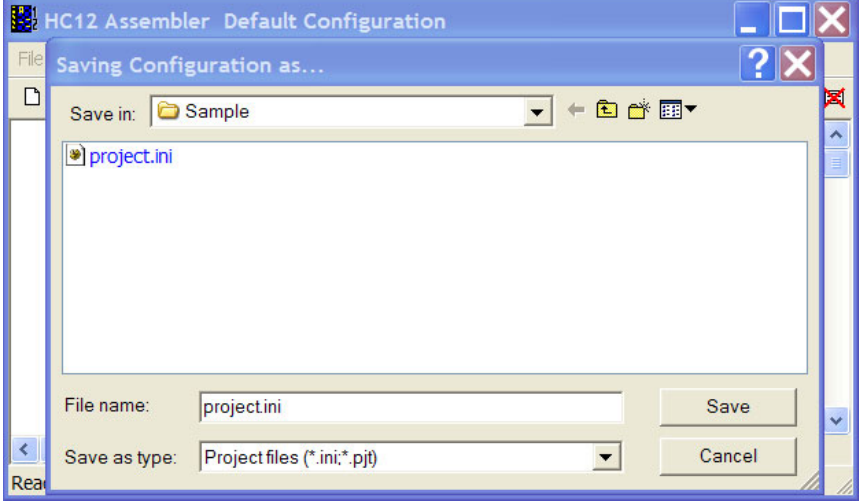
```
; export symbols
            XDEF Entry            ; export 'Entry' symbol
            ABSENTRY Entry        ; for absolute assembly: Mark this
                                  ; as the application entry point.

; include derivative specific macros - RAMStart and RAMEnd data
            INCLUDE 'mc9s12c32.inc'

ROMStart    EQU   $4000 ; absolute address to place my code/constants

; variable/data section
            ORG RAMStart          ; Insert here your data definition.
Counter     DS.W  1
FiboRes     DS.W  1


; code section
            ORG   ROMStart
Entry:
            LDS   #RAMEnd+1       ; initialize the stack pointer to
                                  ; highest absolute RAM address
            CLI                   ; enable interrupts
mainLoop:
            LDX   #1              ; X contains counter
counterLoop:
            STX   Counter         ; update global.
            BSR   CalcFibo
            STD   FiboRes         ; store result
            LDX   Counter
            INX
            CPX   #24             ; larger values cause overflow.
            BNE   counterLoop
            BRA   mainLoop        ; restart.

CalcFibo: ; Function to calculate Fibonacci numbers. Argument is in X
            LDY   #$00            ; second last
            LDD   #$01            ; last
            DBEQ  X,FiboDone      ; loop once more (if X was 1, were
FiboLoop:
            LEAY  D,Y             ; overwrite second last with new va
            EXG   D,Y             ; exchange them -> order is correct
            DBNE  X,FiboLoop
FiboDone:
            RTS                   ; result in D


;***************************************************************
```

```
;*                    Interrupt Vectors                              *
;********************************************************************
             ORG    $FFFE
             DC.W   Entry            ; Reset Vector
```

Store the absolute-assembly form of `main.asm` in a new project directory.

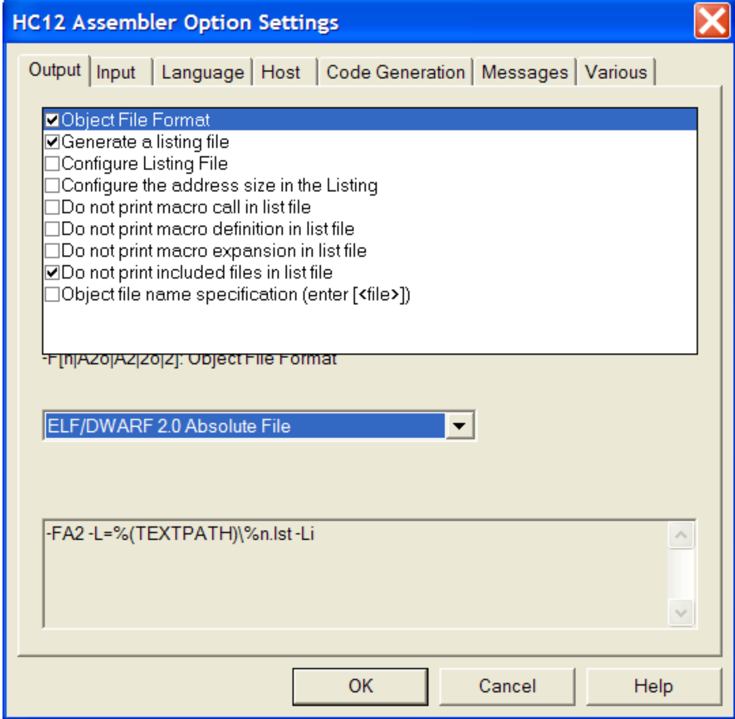
- Start the Assembler. You can do this by opening the `ahc12.exe` file in the `prog` folder in the HC(S)12 CodeWarrior installation. The Assembler opens. Close the *Tip of the Day* dialog box if this dialog box is open.
- Create a new `project.ini` configuration file (*File > New / Default Configuration* and store it in the project directory (*File > Save Configuration As...*. This makes the project directory the current directory for the Assembler (Figure 1.35 on page 67).

Figure 1.35 Creating a new absolute assembly project



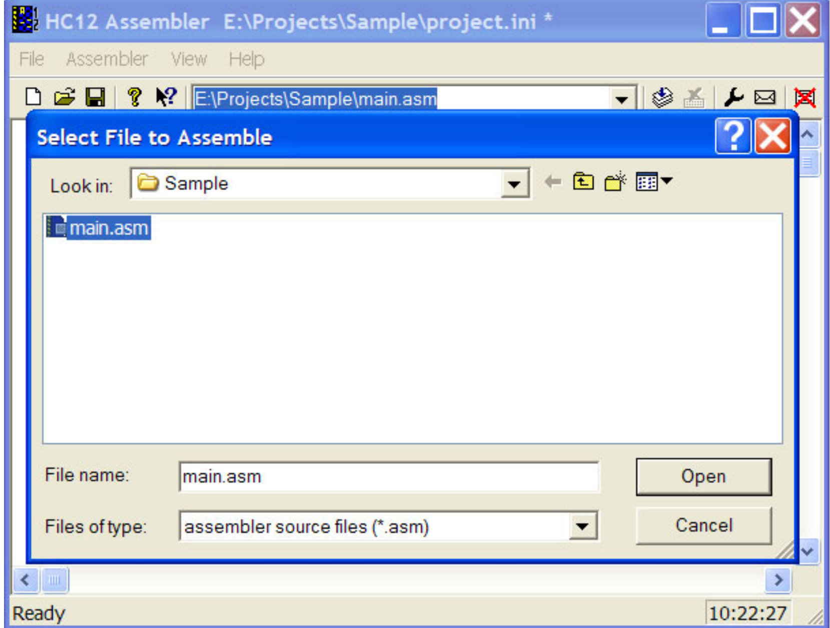
- Select *Assembler > Options*. The *HC12 Assembler Option Settings* dialog box appears (Figure 1.36 on page 68).

Figure 1.36 HC12 Assembler Option Settings dialog box



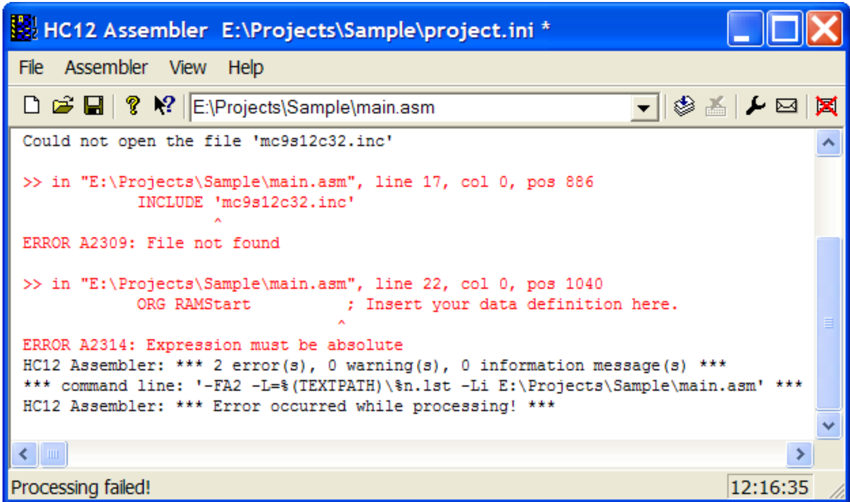
- In the *Output* panel, select the check box in front of *Object File Format*. The Assembler displays more information at the bottom of the dialog box. Select the *ELF/DWARF 2.0 Absolute File* radio button. The assembler options for generating a listing file can also be set at this point, if desired. Click *OK*.
- Select the assembly source-code file that will be assembled: *Select File* > *Assemble*. The *Select File to Assemble* dialog box appears (Figure 1.37 on page 69).

Figure 1.37 Select File to Assemble dialog box



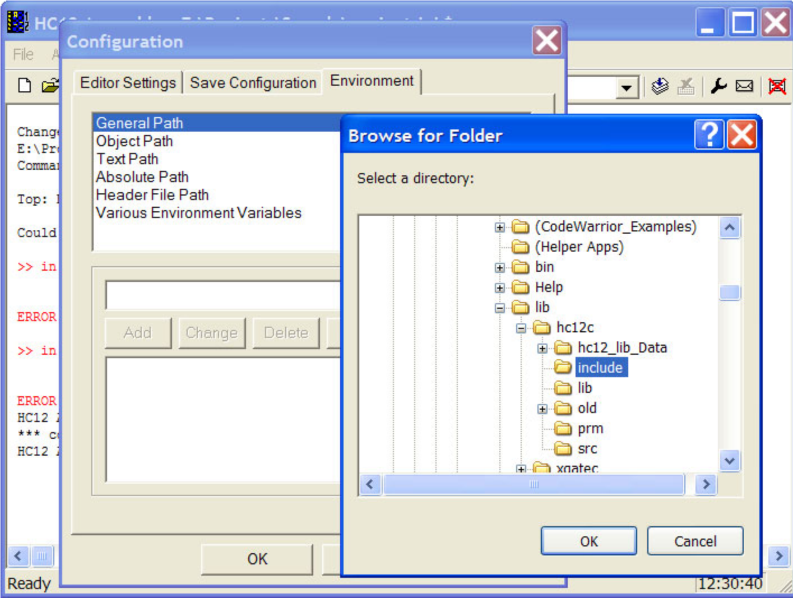
- Browse to the assembly source-code file. Click *Open*. The Assembler now assembles the source code. Error-message (Figure 1.38 on page 70) or positive feedback about the assembly process is created in the assembler main window.

Figure 1.38 “ERROR A2309: File not found” error message



Make sure that the `GENPATH` configuration is set (Figure 1.39 on page 71) for the include file used by the `main.asm` file in this project in the event an error message for a missing file appears, as above.

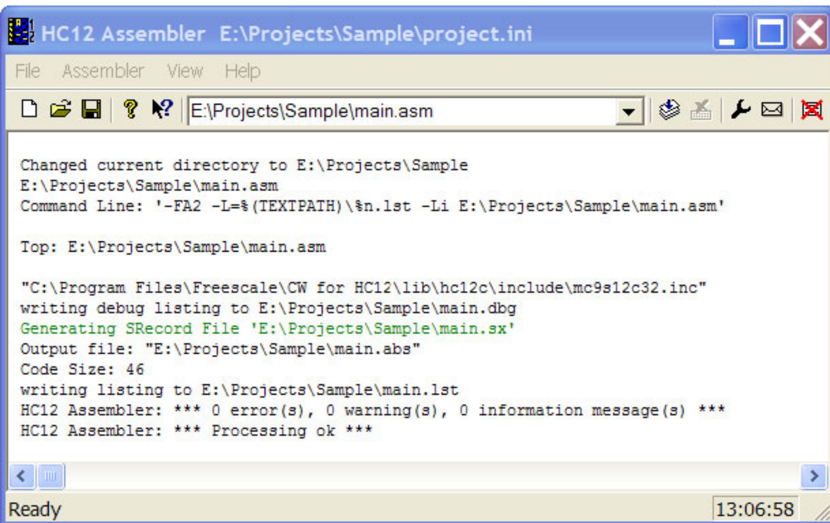
Figure 1.39 Adding a GENPATH for the include file



Confer “Adding a GENPATH” on page 48 for instructions for setting a GENPATH. (*File* > *Configuration...* > *Environment* > *General Path* and browse for the missing include file.) After setting a GENPATH to the folder of the include file, try assembling again.

Select File > *Assemble* and browse for the `*.asm` file and press *Open* for the assembly command. This time, it should assemble correctly (Figure 1.40 on page 72).

Figure 1.40 Successful absolute assembly



The messages indicate that:

- An assembly source code (`main.asm`) file and an `MC68HC908GP32.inc` file were read as input.
- A debugging (`main.dbg`) file was generated in the project directory.
- An S-Record File was created, `main.sx`. This file can be used to program ROM memory.
- An absolute executable file was generated, `main.abs`.
- The Code Size was 46 bytes.

The `main.abs` file can also be used as input to the Simulator/Debugger - another Build Tool in the CodeWarrior Development Studio, with which you can follow the execution of your program.

2

Assembler Graphical User Interface

The Macro Assembler runs under *Windows 9X*, *Windows NT, 2000, XP, 2003, and compatible operating systems*.

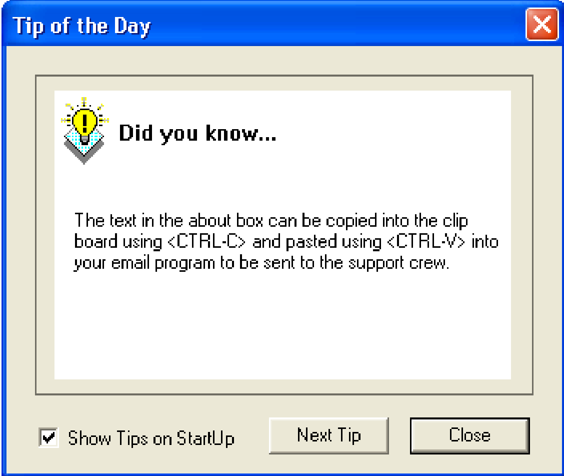
This chapter covers the following topics:

- Starting the Assembler on page 73
- Assembler main window on page 74
- Editor Settings dialog box on page 81
- Save Configuration dialog box on page 87
- Option Settings dialog box on page 89
- Message Settings dialog box on page 89
- About... dialog box on page 92
- Specifying the input file on page 92
- Message/Error feedback on page 93

Starting the Assembler

When you start the Assembler, the Assembler displays a standard *Tip of the Day* (Figure 2.1 on page 74) window containing news and tips about the Assembler.

Figure 2.1 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* window when the Assembler is started, uncheck *Show Tips on StartUp*.

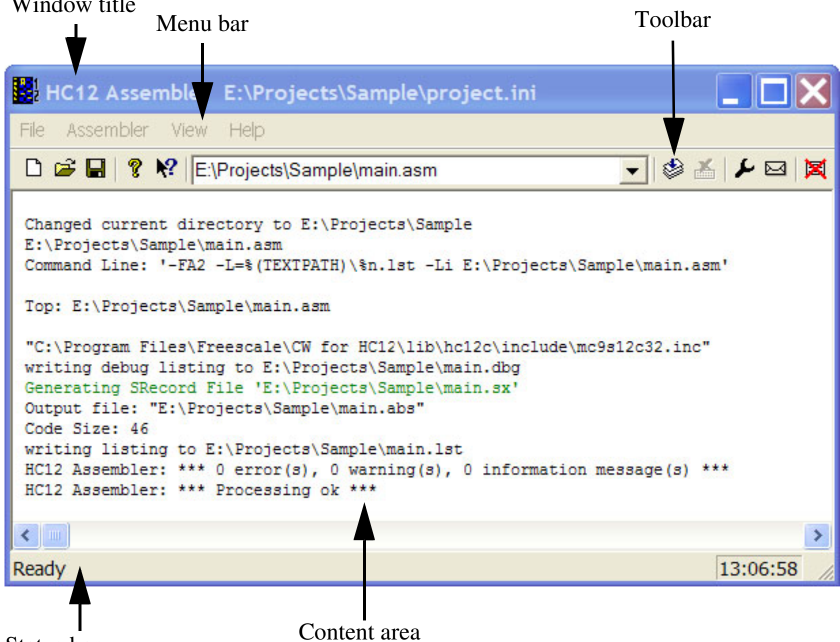
If you want the Assembler to automatically open the standard *Tip of the Day* window at Assembler start up, choose *Help* > *Tip of the Day*.... The Assembler displays the *Tip of the Day* dialog box. Check the *Show Tips on StartUp* check box.

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 (Figure 2.2 on page 75).

Figure 2.2 Assembler main window



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.

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.

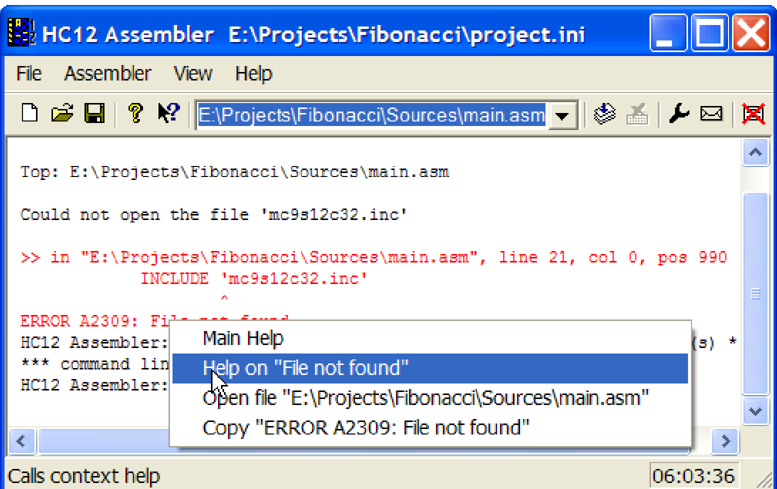
All text in the assembler window content area can have context information consisting of two items:

- 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 (Figure 2.3 on page 76).

Figure 2.3 Right-context Help



If the Assembler cannot open a file even though a context menu entry is present, then the editor configuration information is incorrect (see the on page 81Editor Settings dialog box on page 81 section below).

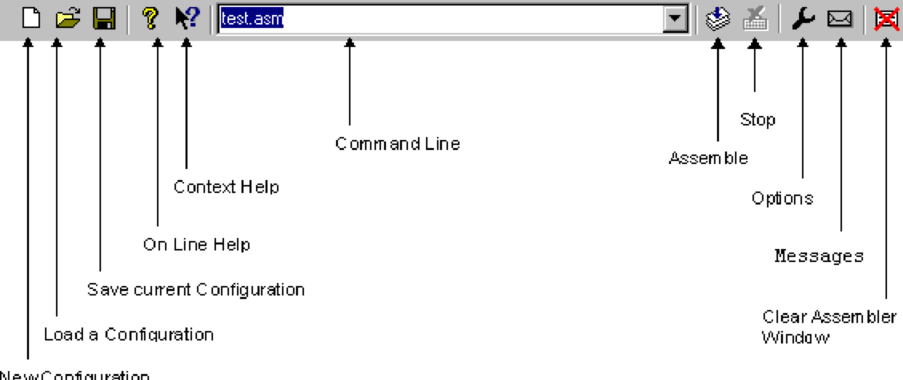
The message number is available for any message output. There are three ways to open the corresponding entry in the help file:

- 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.

Toolbar

Figure 2.4 on page 77 displays the elements of the Toolbar.

Figure 2.4 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.

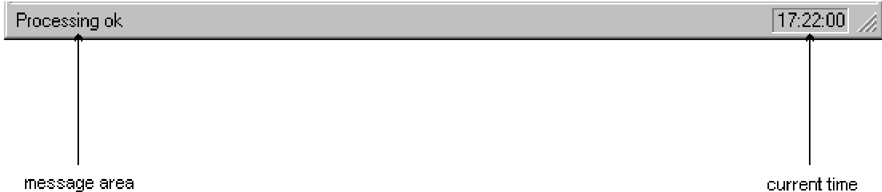
Pressing the *Message Dialog Box* button opens the *Message Settings* dialog box.

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

Status bar

Figure 2.5 on page 78 displays the elements of the Status bar.

Figure 2.5 Status bar



When pointing to a button in the tool bar or a menu entry, the message area displays the function of the button or menu entry to which you are pointing.

Assembler menu bar

The following menus are available in the menu bar (Table 2.1 on page 78):

Table 2.1 Menu bar options

Menu	Description
File menu on page 78	Contains entries to manage Assembler configuration files
Assembler menu on page 80	Contains entries to set Assembler options
View menu on page 80	Contains entries to customize the Assembler window output
Help	A standard Windows Help menu

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 Editor Settings dialog box on page 81.),
- 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 2.2 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 on page 123 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*.

Table 2.2 File menu options (*continued*)

Menu entry	Description
Configuration...	Opens the *Configuration* dialog box to specify the editor used for error feedback and which parts to save with a configuration. See Editor Settings 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.

Assembler menu

The Assembler menu (Table 2.3 on page 80) allows you to customize the Assembler. You can graphically set or reset the Assembler options or to stop the assembling process.

Table 2.3 Assembler menu options

Menu entry	Description
Options	Defines the options which must be activated when assembling an input file. (See Option Settings dialog box on page 89)
Messages	Maps messages to a different message class (See Message Settings dialog box on page 89)
Stop assembling	Stops the assembling of the current source file.

View menu

The View menu (Table 2.4 on page 80) 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.

Table 2.4 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.

Table 2.4 View menu options (*continued*)

Menu entry	Description
Log...	Customizes the output in the assembler window content area. The following two entries in this table are available when Log... is selected:
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.

Editor Settings dialog box

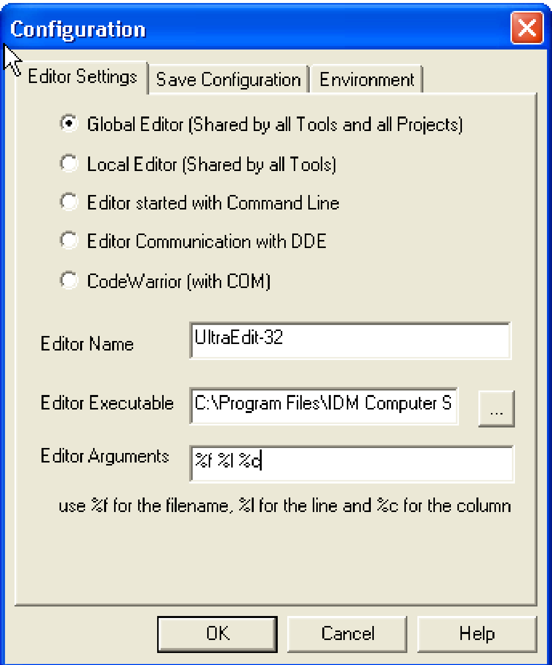
The Editor Setting dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

These are the following main entries:

Global Editor (shared by all tools and projects)

This entry (Figure 2.6 on page 82) is shared by all tools (Compiler/Linker/Assembler/...) for all projects. This setting is stored in the `[Editor]` section of the `mcutools.ini` global initialization file. Some Modifiers on page 86 can be specified in the editor command line.

Figure 2.6 Global Editor Configuration

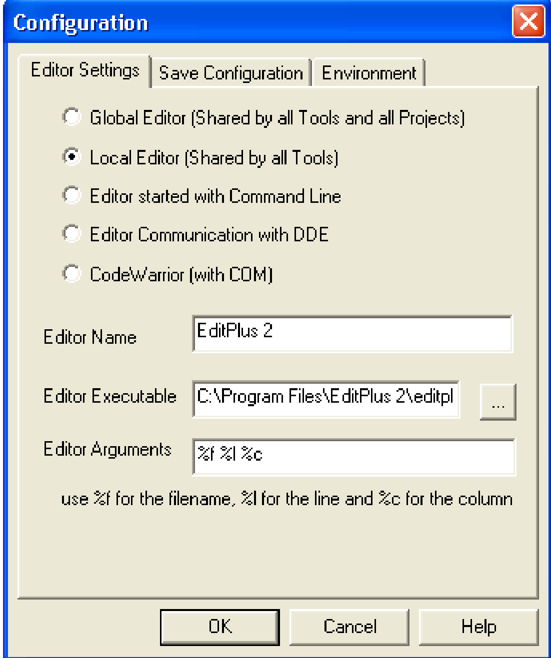


Local Editor (shared by all tools)

This entry (Figure 2.7 on page 83) is shared by all tools (Compiler/Linker/Assembler/...) 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 on page 86 can be specified in the editor command line.

The global or local editor configuration affects other tools besides the Assembler. It is recommended to close other tools while modifying these topics.

Figure 2.7 Local Editor configuration



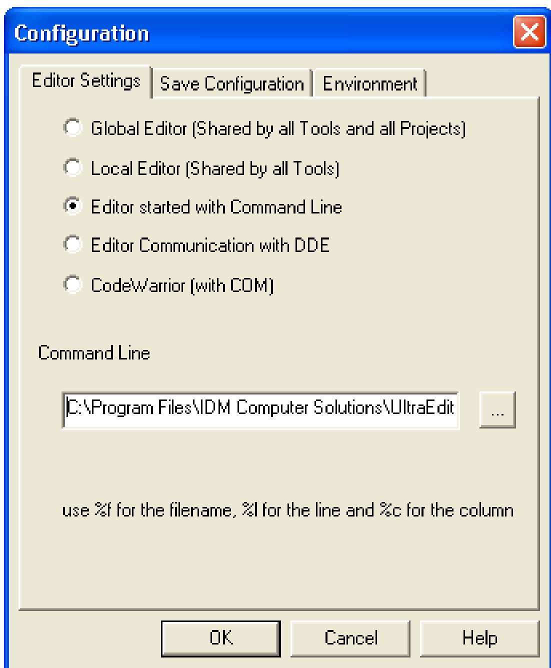
Editor started with the command line

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.

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 on page 86 section below.)

Figure 2.8 Command-Line Editor configuration



Examples of configuring a command-line editor

The following cases portray the syntax used for configuring two external editors. Listing 2.1 on page 84 can be used for the *CodeWright* editor (with an adapted path to the `cw32.exe` file). For *WinEdit* 32 bit version, use the configuration in Listing 2.2 on page 84 (with an adapted path to the `winedit.exe` file).

Listing 2.1 CodeWright editor configuration

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

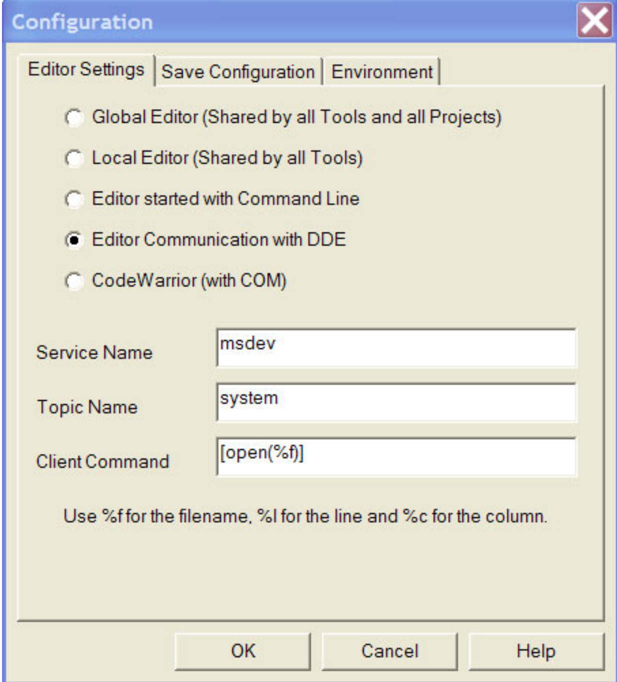
Listing 2.2 WinEdit editor configuration

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

Editor started with DDE

Enter the service, topic and client name to be used for a DDE (Dynamic Data Exchange) connection to the editor. All entries can have modifiers for the filename and line number, as explained in the Modifiers on page 86 section. See Figure 2.9 on page 85.

Figure 2.9 DDE Editor configuration



For the Microsoft Developer Studio, use the following settings (Listing 2.3 on page 85):

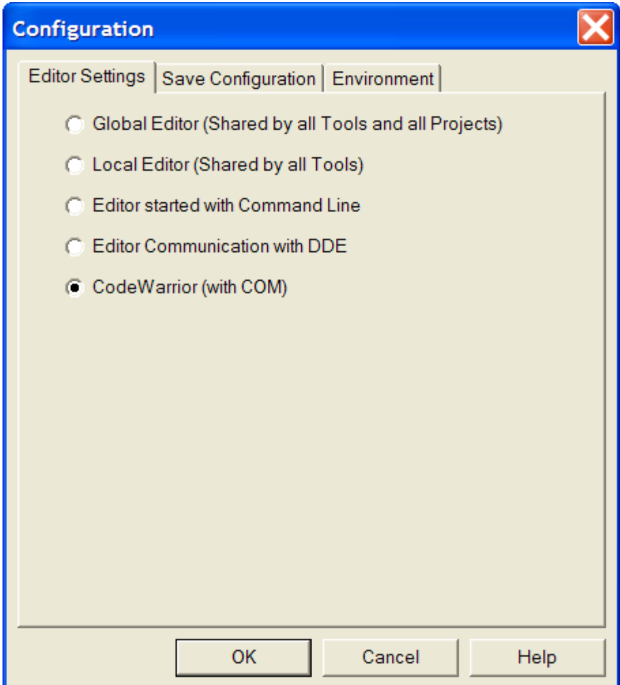
Listing 2.3 Microsoft Developer Studio configuration settings

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

CodeWarrior with COM

If CodeWarrior with COM is enabled, the CodeWarrior IDE (registered as a COM server by the installation script) is used as the editor (Figure 2.10 on page 86).

Figure 2.10 COM Editor Configuration



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 Be careful. 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`
Please check your editor's manual to define the command line which should be used to start the editor.

Save Configuration dialog box

The second index of the configuration dialog box contains all options for the save operation.

In the *Save Configuration* index, there are four check boxes where you can choose 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.

NOTE 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 on page 185 assembler option at startup or loading the configuration manually, a different name other than `project.ini` can be chosen.

Environment Configuration dialog box

The third page of the dialog is used to configure the environment. The content of the dialog is read from the actual project file out of the [Environment Variables] section.

The following variables are available:

- General Path: `GENPATH`
- Object Path: `OBJPATH`
- Text Path: `TEXTPATH`
- Absolute Path: `ABSPATH`
- Header File Path: `LIBPATH`

Various Environment Variables: other variables not covered by the above list.

The following buttons are available:

- 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.

Option Settings dialog box

This dialog box allows you to set/reset assembler options. 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 2.5 on page 89):

Table 2.5 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++, ...)
Host	Lists options related to the host.
Code Generation	Lists options related to code generation (memory models, ...).
Messages	Lists options controlling the generation of error messages.
Various	Lists various additional options (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 on page 123 chapter.

Message Settings dialog box

You can use the Message Settings dialog box to map messages to a different message class.

Some buttons in the dialog box may be disabled. For example, if an option cannot be moved to an information message, the '*Move to: Information*' button is disabled. The following buttons are available in the dialog box (Table 2.6 on page 90):

Table 2.6 Message Settings options

Button	Description
Move to: Disabled	The selected messages are disabled; they will no longer be displayed.
Move to: Information	The selected messages are changed to information messages.
Move to: Warning	The selected messages are changed to warning messages.
Move to: Error	The selected messages are changed to error messages.
Move to: Default	The selected messages are changed to their default message 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.

A panel is available for each error message class and the content of the list box depends on the selected panel (Table 2.7 on page 90):

Table 2.7 Types of message groups

Message group	Description
Disabled	Lists all disabled messages. That means that messages displayed in the list box will not be displayed by the Assembler.
Information	Lists all information messages. Information messages informs about action taken by the Assembler.
Warning	Lists all warning messages. When such a message is generated, translation of the input file continues and an object file will be generated.

Table 2.7 Types of message groups (*continued*)

Message group	Description
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.

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 list box and then click the button associated with the class where you want to move the message.

Example:

To define the warning ‘*A2336: Value too big*' as an error message:

- Click the *Warning* sheet to display the list of all warning messages in the list box.
- Click on the string ‘*A2336: Value too big*' in the list box to select the message.
- Click *Error* to define this message as an error message.

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.

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 on *OK* to close this dialog box.

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

Specifying the 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.

Use the command line in the toolbar to assemble

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

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 press the *Assemble* button in the tool bar or when you press the enter key.

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 tool bar is selected.

Use the 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.

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.

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 Listing 2.4 on page 93.

Listing 2.4 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 Listing 2.5 on page 94.

Listing 2.5 Error message example

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

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

For different message formats, see the following Assembler options:

- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203,
- -WmsgFob: Message format for batch mode on page 205,
- -WmsgFoi: Message format for interactive mode on page 207,
- -WmsgFonf: Message format for no file information on page 209, and
- -WmsgFonp: Message format for no position information on page 211.

Use information from the 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.

Use a 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.

Line number can be specified on the command line

Editors like *UltraEdit-32*, *WinEdit* (*v95* or higher), or *CodeWright* 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.

Line number cannot be specified on the command line

Editors like *WinEdit v31* 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:

- Activate the assembler again.
- Click the line on which the message was generated. This line is highlighted on the screen.
- Copy the line in the clipboard by pressing `CTRL + C`.
- Activate the editor again.
- Select *Search* > *Find*; the standard *Find* dialog box is opened.
- Paste the contents of the clipboard in the Edit box pressing `CTRL + V`.
- Click *Forward* to jump to the position where the error was detected.

3

Environment

This part describes the environment variables used by the Assembler. Some of those 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:

1) The current project file with the Environment Variables section. This file may be specified on Tool startup using the -Prod: Specify project file at startup on page 185 assembler option. This is the recommended method and is also supported by the IDE.

2) 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 on page 108 environment variable. Using the default.env file is not recommended.

3) 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 so-called environment variables. The syntax is always the same (Listing 3.1 on page 97).

Listing 3.1 Syntax for setting environment variables

```
Parameter: KeyName"="ParamDef.
```

Listing 3.2 on page 97 is a typical example of setting an environment variable.

Listing 3.2 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.
- 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 on page 148 assembler option.

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., Make Utility, ...).

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 a Make utility, ...), 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 on page 107 system environment variable may be used.

The current directory is displayed among other information with the -V: Prints the Assembler version on page 187 assembler option and in the *About...* box.

Environment macros

It is possible to use macros (Listing 3.3 on page 99) in your environment settings.

Listing 3.3 Using a macro for setting environment variables

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

In the example in Listing 3.3 on page 99, `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 3.4 on page 99 are allowed. Note that they are case-sensitive and always surrounded by `{}`. Also the variable content contains a directory separator '`\`' as well.

Listing 3.4 Special variables used with macros for setting environment variables

`{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 were your Windows O/S is installed, e.g., '`C:\WINNT\`'.

Global initialization file - mcutools.ini (PC only)

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`).

Listing 3.5 on page 100 shows two typical locations used for the `mcutools.ini` files.

Listing 3.5 Usual locations for the mcutools.ini files

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

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 (Assembler, Linker, etc.).

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 on page 105 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 directory is loaded.

Local Configuration File Entries on page 409 documents the sections and entries you can put in a project.ini file.

Paths

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 in Listing 3.6 on page 101.

Listing 3.6 Syntax used for setting path lists of environment variables

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

Listing 3.7 on page 101 is a typical example of setting an environment variable.

Listing 3.7 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. Listing 3.8 on page 101 shows the use of an asterisk (*) for recursively searching the entire C drive for a configuration file with a \INSTALL\LIB path.

Listing 3.8 Recursive search for a configuration file

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

NOTE Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used. For further details refer to GENPATH=.\; TEXTFILE=.\txt Environment variable details on page 103.

We strongly recommend working with the Shell and setting the environment by means of a default.env file in your project directory. (This 'project dir' can be set in the

Shell's 'Configure' dialog box). Doing it this way, you can have different projects in different directories, each with its own environment.

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

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.

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 '\' (Listing 3.9 on page 102):

Listing 3.9 Using multiple lines for an environment variable

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

Listing 3.9 on page 102 is the same as the alternate source code in Listing 3.10 on page 102.

Listing 3.10 Alternate form of Listing 3.9 on page 102

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

But this feature may be dangerous when used together with paths (Listing 3.11 on page 102).

Listing 3.11 A path is included by the line continuation character

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

will result in

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

In order 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 (Listing 3.12 on page 103).

Listing 3.12 Recommended style whenever a backslash is present

```
GENPATH=.\;
TEXTFILE=.\txt
Environment variable 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 (Table 3.1 on page 103).

Table 3.1 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 default.env for the PC or in the .hidefaults for UNIX.
See also (if needed)	Names related sections.

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
```

ASMOPTIONS: Default assembler options

Tools

Assembler

Syntax

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

Arguments

`<option>`: Assembler command-line option

Description

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

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

Example

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

See also

Assembler Options on page 123

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

Environment variables:

- USERNAME: User Name in object file on page 118
- INCLUDETIME: Creation time in the object file on page 113

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” on page 98

“Global initialization file - mcutools.ini (PC only)” on page 99

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
```

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

Listing 3.13 on page 109 lists all errors into the `MyErrors.err` file in the current directory.

Listing 3.13 Naming an error file

```
ERRORFILE=MyErrors.err
```

Listing 3.14 on page 110 lists all errors into the errors file in the \tmp directory.

Listing 3.14 Naming an error file in a specific directory

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

Listing 3.15 on page 110 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 3.15 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 (Listing 3.16 on page 110).

Listing 3.16 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 (Listing 3.17 on page 110).

Listing 3.17 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.

Another example (Listing 3.18 on page 111) shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called EDOUT:

Listing 3.18 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 Be careful to set this variable if the WinEdit Editor is used, otherwise the editor cannot find the `EDOUT` file.

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
```

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

Environment variables:

- COPYRIGHT: Copyright entry in object file on page 106
- USERNAME: User Name in object file on page 118

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
```

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 S-Record File type. 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
```

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
```

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 on page 98

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

Environment variables:

- COPYRIGHT: Copyright entry in object file on page 106
- INCLUDETIME: Creation time in the object file on page 113

4

Files

This chapter covers:

- "Input files" on page 119
- "Output files" on page 119
- "File Processing" on page 122

Input files

Input files to the Assembler:

- Source files on page 119
- Include files on page 119

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 on page 112

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 on page 107 environment variable.

Output files

Output files from the Assembler:

- Object files on page 120
- Absolute files on page 120
- S-Record Files on page 120

- Listing files on page 121
- Debug listing files on page 121
- Error listing file on page 121

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 on page 114 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.

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 on page 104 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.

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 a S-Record File is generated at the same time. This file can be burnt into a ROM. 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 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.

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 on page 153 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 on page 116.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 on page 355 chapter.

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 on page 114 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.

Error listing file

If the Macro Assembler detects any errors, it does not create an object file but does create an error listing file. This file is generated in the directory the source file was found (see ERRORFILE: Filename specification error on page 109.

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), 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.

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.

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.

File Processing

Figure 4.1 on page 122 shows how the Assembler locates its input and output files.

Figure 4.1 File processing with the Assembler

5

Assembler Options

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 dash/minus `(-)` followed by one or more letters or digits. Anything not starting with a dash/minus 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 on page 105 (Table 5.1 on page 123) 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 coalescing options in the same group, i.e., one might also write `"-Lci"` instead of `"-Lc -Li"`. However such a usage is not recommended as it make 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 coalesce options in different groups, e.g., `"-Lc -W1"` *cannot* be abbreviated by the terms `"-LC1"` or `"-LCW1"`.

Table 5.1 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 (Table 5.2 on page 124) are grouped by:

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

Table 5.2 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++, ...)
Host	Lists options related to the host.
Code Generation	Lists options related to code generation (memory models, ...).
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 (Table 5.3 on page 124).

Table 5.3 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.
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.

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 (Table 5.4 on page 125).

Table 5.4 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 `default.env` for the PC or in the `.hidefaults` for UNIX.
See also (if needed)	Names related options.

Using special modifiers

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

The following modifiers are supported (Table 5.5 on page 125).

Table 5.5 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

Table 5.5 Special modifiers for assembler options (*continued*)

Modifier	Description
%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 ‘%’

Examples using special modifiers

The assumed path and filename (filename base for the modifiers) used for the examples Listing 5.2 on page 126 through Listing 5.13 on page 128 is displayed in Listing 5.1 on page 126.

Listing 5.1 Example filename and path used for the following examples

```
C:\Freescale\my demo\TheWholeThing.myExt
```

Using the `%p` modifier as in Listing 5.2 on page 126 shows the path with a file separator but without the filename.

Listing 5.2 %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 (Listing 5.3 on page 126).

Listing 5.3 %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 (Listing 5.4 on page 127).

Listing 5.4 %n returns just the filename without the file extension

```
TheWholeThing
```

Using `%E` as a modifier returns the first three characters in the file extension (Listing 5.5 on page 127).

Listing 5.5 %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 (Listing 5.6 on page 127).

Listing 5.6 %e is used for returning the whole extension

```
myExt
```

The `%f` modifier returns the path and the filename but without the file extension (Listing 5.7 on page 127).

Listing 5.7 %f gives the path plus the filename (no file extension)

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

The path in Listing 5.1 on page 126 contains a space, therefore using `%"` or `%'` is recommended
(Listing 5.8 on page 127 or Listing 5.9 on page 127).

Listing 5.8 Use %"%f%" in case there is a space in its path, filename, or extension

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

Listing 5.9 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 Listing 5.10 on page 128, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.11 on page 128.

Listing 5.10 Example for setting TEXTPATH

```
TEXTPATH=C:\Freescale\txt
```

Listing 5.11 $(TEXTPATH)\myfile.txt where TEXTPATH is defined

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

However, if `TEXTPATH` does not exist or is empty, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.12 on page 128).

Listing 5.12 $(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 Listing 5.13 on page 128.

Listing 5.13 %% allows a percent sign to be expressed

```
myExt%
```

List of assembler options

The following table lists each command line option you can use with the Assember (Table 5.6 on page 128)

Table 5.6 Assembler options

Assembler option
-C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON on page 132
-Ci: Switch case sensitivity on label names OFF on page 133
-CMacAngBrack: Angle brackets for grouping Macro Arguments on page 135
-CMacBrackets: Square brackets for macro arguments grouping on page 136
-Compat: Compatibility modes on page 137
-CpDirect: Define DIRECT register value on page 140

Table 5.6 Assembler options (*continued*)

Assembler option
-Cpu (-CpuCPU12, -CpuHCS12, -CpuHCS12X): Derivative on page 143
-D: Define Label on page 146
-Env: Set environment variable on page 148
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output-file format on page 149
-H: Short Help on page 151
-I: Include file path on page 152
-L: Generate a listing file on page 153
-Lasmc: Configure listing file on page 156
-Lasms: Configure the address size in the listing file on page 158
-Lc: No Macro call in listing file on page 160
-Ld: No macro definition in listing file on page 163
-Le: No Macro expansion in listing file on page 166
-Li: Not included file in listing file on page 169
-Lic: License information on page 171
-LicA: License information about every feature in directory on page 172
-LicBorrow: Borrow license feature on page 173
-LicWait: Wait until floating license is available from floating License Server on page 175
-MacroNest: Configure maximum macro nesting on page 177
-M (-Ms, -Mb, -Ml): Memory Model on page 176
-MCUasm: Switch compatibility with MCUasm ON on page 178
-N: Display notify box on page 179
-NoBeep: No beep in case of an error on page 180
-NoDebugInfo: No debug information for ELF/DWARF files on page 181
-NoEnv: Do not use environment on page 182
-ObjN: Object filename specification on page 183

Table 5.6 Assembler options (*continued*)

Assembler option
-Prod: Specify project file at startup on page 185
-Struct: Support for structured types on page 186
-V: Prints the Assembler version on page 187
-View: Application standard occurrence on page 188
-W1: No information messages on page 190
-W2: No information and warning messages on page 191
-WErrFile: Create "err.log" error file on page 192
-Wmsg8x3: Cut filenames in Microsoft format to 8.3 on page 193
-WmsgCE: RGB color for error messages on page 195
-WmsgCF: RGB color for fatal messages on page 196
-WmsgCI: RGB color for information messages on page 197
-WmsgCU: RGB color for user messages on page 198
-WmsgCW: RGB color for warning messages on page 199
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
-WmsgFob: Message format for batch mode on page 205
-WmsgFoi: Message format for interactive mode on page 207
-WmsgFonf: Message format for no file information on page 209
-WmsgFonp: Message format for no position information on page 211
-WmsgNe: Number of error messages on page 213
-WmsgNi: Number of Information messages on page 214
-WmsgNu: Disable user messages on page 215
-WmsgNw: Number of Warning messages on page 217
-WmsgSd: Setting a message to disable on page 218

Table 5.6 Assembler options (*continued*)

Assembler option
-WmsgSe: Setting a message to Error on page 219
-WmsgSi: Setting a message to Information on page 220
-WmsgSw: Setting a Message to Warning on page 221
-WOutFile: Create error listing file on page 222
-WStdout: Write to standard output on page 223

Detailed listing of all assembler options

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

-C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON

Group

Various

Scope

Assembly Unit

Syntax

```
-C=SAvocet
```

Arguments

None

Default

none.

Description

This switches ON compatibility mode with the Avocet Assembler. Additional features supported when this option is activated are enumerated in the "Semi-Avocet Compatibility" on page 435.

Example

```
ASMOPTIONS=-C=SAvocet
```

See also

Semi-Avocet Compatibility on page 435

-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.

Example

When case sensitivity on label names is switched off, the Assembler will not generate an error message for the assembly source code in Listing 5.14 on page 134.

Listing 5.14 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 on page 149

-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. New code should use the `[? ?]` syntax instead.

See also

Macro argument grouping on page 349

-CMacBrackets: Square brackets for macro arguments grouping on page 136

-CMacBrackets: Square brackets for macro arguments grouping

Group

Language

Scope

Application

Syntax

`-CMacBrackets(ON|OFF)`

Arguments

`ON` or `OFF`

Default

`ON`

Description

This option control 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.

See also

Macro argument grouping on page 349

-CMacAngBrack: Angle brackets for grouping Macro Arguments on page 135

-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

NOTE 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 issues a warning if such a comment does not start with a `"*"` or a `";"`.

Examples

– Listing 5.15 on page 138 demonstrates that when `-Compat=c`, comments can start with an asterisk, `*`.

Listing 5.15 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 Listing 5.16 on page 138 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 5.16 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 the line-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 on page 338 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 on page 307 assembly directive to generate repeated patterns more easily without having to use recursive macros.

-CpDirect: Define DIRECT register value

Group

Code Generation

Scope

Application

Syntax

```
-CpDirect<num>
```

`<num>` is the start address of the memory window.

Arguments

`<num>`

Default

```
-CpDirect0x0000
```

Description

For the HC12 or HCS12 families, all direct accesses were accessing the address range from 0x0000 to 0x00FF. In this range, a resource which is frequently used could be mapped to benefit from the shorter direct-addressing mode compared to the extended- addressing mode.

For the HCS12X, the mapping of RAM, registers, and EEPROM is no longer supported. Instead, the direct accesses can now be configured to map to any boundary in memory which is a multiple of 256 bytes.

Because of this change, the Assembler does need to know which part of the address space is accessible through with the direct-addressing mode.

With the `-CpDirect0` assembler option, the generated code is as for the HC12 or HCS12.

NOTE This optimization is only useful for if the address is known. Variables allocated in a `SHORT` section are not affected by this option.

Example

Consider the following code in Listing 5.17 on page 141:

Listing 5.17 Example assembly code

```
        ORG $50
data:   DS.B 1

MyCode: SECTION
Entry:
        LDS   #$AFE               ; init Stack Pointer
        LDAA  #$01
main:   STAA  data
        STAA $1150
        BRA main
```

By default, or with `-CpDirect0x0000` option, the following assembler listing is generated (Listing 5.18 on page 141):

Listing 5.18 Default assembler output listing or when using the -CpDirect0x0000 option

```
                                   ORG   $50
a000050                data:       DS.B  1

                       MyCode:     SECTION
                       Entry:
 000000 CF0A FE                    LDS   #$AFE ; init Stack Pointer
 000003 8601                       LDAA  #$01
 000005 5A50           main:       STAA  data
 000007 7A11 50                    STAA  $1150
 00000A 20F9                       BRA   main
```

When using the -CpDirect0x1100 option (with the DIRECT page register contains 0x11), the assembler output listing (Listing 5.19 on page 141) is generated.

Listing 5.19 Assembler output listing when using the -CpDirect0x1100 option

```
                                   ORG   $50
 a000050                data:      DS.B  1

                        MyCode:    SECTION
                        Entry:
  000000 CF0A FE                   LDS   #$AFE ; init Stack Pointer
  000003 8601                      LDAA  #$01
```

```
000005 7A00 50     main:      STAA  data
000008 5A50                   STAA  $1150
00000A 20F9                   BRA   main
```

-Cpu (-CpuCPU12, -CpuHCS12, -CpuHCS12X): Derivative

Group

Code Generation

Scope

Application

Syntax

`-Cpu{CPU12|HCS12|HCS12X}`

Arguments

None

Default

`-CpuCPU12`

Description

This option controls whether code for an HC12, an HCS12, or for an HCS12X is produced.

The instruction formats for the CPU12 and the HCS12 are very similar; these two options do only differ in the PCR-relative `MOVB/MOVW` instructions.

In the CPU12 (or default) mode, the Assembler adapts the offsets according to the CPU12 Reference Manual, paragraph 3.9.1 Move Instructions. In the HCS12 mode, it does not.

In the HCS12X mode, the Assembler supports the additional HCS12X instructions. For the `MOVB` and `MOVW` instructions, it also supports their additional addressing modes.

Examples

Consider the source code in Listing 5.20 on page 144:

Listing 5.20 Example assembly code

```
One:     DC 1
CopyOne: MOVB One, PCR, $1000
```

Using the default or with -CpuCPU12 assembler option, the Assembler generates the output listing in Listing 5.21 on page 144:

Listing 5.21 Assembler output listing when using the default or the -CpuCPU12 option

```
000000 01          One:     DC 1
000001 180D DC10   CopyOne: MOVB One, PCR, $1000
003005 00
```

With the `-CpuHCS12` or the `-CpuHCS12X` option, the Assembler generates the output listing in Listing 5.22 on page 144:

Listing 5.22 Assembler output listing when using the -CpuHCS12 or the -CpuHCS12X option

```
003000 01          One:     DC 1
003001 180D DA10   CopyOne: MOVB One, PCR, $1000
003005 00
```

The difference is that for the CPU12 the Assembler adapts the offset to `One` according to the `MOVB IDX/EXT` case by -2, so the resulting code is `$DC` for the `IDX` encoding. For the HCS12, this is not done, so the `IDX` encodes it as `$DA`.

NOTE PC-relative `MOVB/MOVW` instructions (e.g., "`MOVB 1,PC,2,PC`") are not adapted. Only PCR-relative move instructions (`MOVB 1,PCR,2,PCR`) are adapted.

To assemble HCS12X code, specify the `-CpuHCS12X` option.

Consider the source code in Listing 5.23 on page 145:

Listing 5.23 Example assembly code

```
GLDAA $1234
MOVB $1234,X,$5678,Y
ANDX $CDEF
```

When using the -CpuHCS12X option, the Assembler generates the output listing in Listing 5.24 on page 145:

Listing 5.24 Assembler output listing when using the -CpuHCS12X option

```
    1     1    000000 18B6 1234     GLDAA $1234
    2     2    000004 180A E212     MOVB $1234,X,$5678,Y
               000008 34EA 5678
    3     3    00000C 18B4 CDEF     ANDX $CDEF
```

See also

CPU12 Reference Manual, paragraph 3.9.1 Move Instructions

-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`" would be 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 Listing 5.25 on page 146 and Listing 5.26 on page 147.

Listing 5.25 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 " 2005-"
  YearAsString YEAR
endif
  DC.B 0
endif
```

When assembled with the "`-dADD_COPYRIGHT -dYEAR=2005`" option, the assembler output listing file in Listing 5.26 on page 147 is generated:

Listing 5.26 Generated listing 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 " 2005-"
           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
```

-Env: Set environment variable

Group

Host

Scope

Assembly Unit

Syntax

`-Env<EnvironmentVariable>=<VariableSetting>`

Arguments

`<EnvironmentVariable>`: Environment variable to be set

`<VariableSetting>`: Setting of the environment variable

Default

None

Description

This option sets an environment variable.

Example

```
ASMOPTIONS=-EnvOBJPATH=\sources\obj
```

This is the same as:

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

in the `default.env` file.

See also

“GENPATH=.\; TEXTFILE=.\txt Environment variable details” on page 103

-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.

Use the `-Fh` option to use 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
```

-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

Listing 5.27 on page 151 is a portion of the list produced by the `-H` option:

Listing 5.27 Example help list

```
...
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
...
```

-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
```

-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" on page 125).

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 (Listing 5.28 on page 154), 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 Listing 5.28 on page 154, together with the code from an include file (Listing 5.29 on page 154) generates the output listing in Listing 5.30 on page 154.

Listing 5.28 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 5.29 Example source code from an include file

```
cpChar: MACRO
        LDAA \1
        STAA \2
        ENDM
```

Listing 5.30 Assembly output listing

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
    1    1                              XDEF  Start
    2    2                     MyData:  SECTION
    3    3   000000            char1:   DS.B  1
    4    4   000001            char2:   DS.B  1
    5    5                              INCLUDE "macro.inc"
    6    1i                    cpChar:  MACRO
    7    2i                             LDAA  \1
    8    3i                             STAA  \2
    9    4i                             ENDM
   10    6                     CodeSec: SECTION
   11    7                     Start:
   12    8                              cpChar char1, char2
   13    2m  000000 B6 xxxx   +         LDAA  char1
   14    3m  000003 7A xxxx   +         STAA  char2
   15    9   000006 A7                  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 on page 355 chapter.

See also

Assembler options:

- -Lasmc: Configure listing file on page 156
- -Lasms: Configure the address size in the listing file on page 158
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

-Lasmc: Configure listing file

Group

Output

Scope

Assembly unit

Syntax

`-Lasmc={s|r|m|l|k|i|c|a}`

Arguments

`s` - Do not write the source column

`r` - Do not write the relative column (Rel.)

`m` - Do not write the macro mark

`l` - Do not write the address (Loc)

`k` - Do not write the location type

`i` - Do not write the include mark column

`c` - Do not write the object code

`a` - Do not write the absolute column (Abs.)

Default

Write all columns.

Description

The default-configured listing file shows a lot of information. With this option, the output can be reduced to columns which are of interest. This option configures which columns are printed in a listing file. To configure which lines to print, see the -Lc: No Macro call in listing file on page 160, -Ld: No macro definition in listing file on page 163, -Le: No Macro expansion in listing file on page 166, and -Li: Not included file in listing file on page 169 assembler options.

Examples

For the following assembly source code, the Assembler generates the default-configured output listing (Listing 5.31 on page 157):

```
DC.B   "Hello World"
DC.B   0
```

Listing 5.31 Example assembler output listing

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1    000000 4865 6C6C       DC.B "Hello World"
              000004 6F20 576F
              000008 726C 64
    2    2    00000B 00              DC.B 0
```

In order to get this output without the source file line numbers and other irrelevant parts for this simple `DC.B` example, the following option is added: `"-Lasmc=ramki"`. This generates the output listing in Listing 5.32 on page 157:

Listing 5.32 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 on page 355 chapter.

See also

Assembler options:

- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

- -Lasms: Configure the address size in the listing file on page 158

-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 on page 156 option. To configure which lines to print, see the following assembler options:

- -Lc: No Macro call in listing file on page 160,
- -Ld: No macro definition in listing file on page 163,
- -Le: No Macro expansion in listing file on page 166, and
- -Li: Not included file in listing file on page 169.

Example

For the following instruction:

```
NOP
```

the Assembler generates this default-configured output listing (Listing 5.33 on page 159):

Listing 5.33 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 5.34 on page 159).

Listing 5.34 Example assembler output listing configured with -Lasms1

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   00     XX                 NOP
```

See also

Assembler Listing File on page 355 chapter

Assembler options:

- -Lasmc: Configure listing file on page 156
- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

-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 Listing 5.35 on page 160:

Listing 5.35 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
```

along with additional source code (Listing 5.36 on page 161) from the `macro.inc` include file generates the following output in the assembly listing file (Listing 5.37 on page 161):

Listing 5.36 Example source code from the macro.inc file

```
cpChar:  MACRO
         LDAA  \1
         STAA  \2
         ENDM
```

Listing 5.37 Output assembly listing

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF  Start
    2    2                      MyData:  SECTION
    3    3   000000             char1:   DS.B  1
    4    4   000001             char2:   DS.B  1
    5    5                               INCLUDE "macro.inc"
    6    1i                     cpChar:  MACRO
    7    2i                              LDAA  \1
    8    3i                              STAA  \2
    9    4i                              ENDM
   10    6                      CodeSec: SECTION
   11    7                      Start:
   13    2m  000000 B6  xxxx    +        LDAA char1
   14    3m  000003 B7  xxxx    +        STAA char2
   15    9   000006 01                   NOP
   13    2m  000000 B6  xxxx   +         LDAA  char1
   14    3m  000003 7A  xxxx   +         STAA  char2
   15    9   000006 A7                   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 on page 355 chapter.

See also

Assembler options:

- -L: Generate a listing file on page 153
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

-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 Listing 5.38 on page 163 along with additional source code (Listing 5.39 on page 164) from the `macro.inc` file generates an assembler output listing (Listing 5.40 on page 164) file:

Listing 5.38 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 5.39 Example source code from an include file

```
cpChar:  MACRO
           LDAA   \1
           STAA   \2
         ENDM
```

Listing 5.40 Example assembler output listing

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF  Start
    2    2                      MyData:  SECTION
    3    3   000000             char1:   DS.B  1
    4    4   000001             char2:   DS.B  1
    5    5                               INCLUDE "macro.inc"
    6   1i                      cpChar:  MACRO
   10    6                      CodeSec: SECTION
   11    7                      Start:
   12    8                               cpChar char1, char2
   13   2m   000000 B6 xxxx    +         LDAA  char1
   14   3m   000003 7A xxxx    +         STAA  char2
   15    9   000006 A7                   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 on page 355 chapter.

See also

Assembler options:

- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

-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 Listing 5.41 on page 166 along with additional source code (Listing 5.42 on page 167) from the `macro.inc` file generates an assembly output listing file (Listing 5.43 on page 167):

Listing 5.41 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 5.42 Example source code from an included file

```
cpChar:  MACRO
           LDAA   \1
           STAA   \2
         ENDM
```

Listing 5.43 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                                LDAA   \1
    8   3i                                STAA   \2
    9   4i                                ENDM
   10    6                      CodeSec:  SECTION
   11    7                      Start:
   12    8                                cpChar char1, char2
   15    9   000006 A7                    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 on page 355 chapter.

See also

Assembler options:

- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Li: Not included file in listing file on page 169

-Li: Not 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 Listing 5.44 on page 169 along with additional source code (Listing 5.45 on page 170) from the `macro.inc` file generates the following output in the assembly listing file:

Listing 5.44 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 5.45 Example source code in an include file

```
cpChar:  MACRO
           LDAA   \1
           STAA   \2
         ENDM
```

Listing 5.46 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 B6 xxxx   +          LDAA  char1
   14   3m   000003 7A xxxx   +          STAA  char2
   15    9   000006 A7                   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 on page 355 chapter.

See also

Assembler options:

- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166

-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...* dialog box.

Example

```
ASMOPTIONS=-Lic
```

See also

Assembler options:

- -LicA: License information about every feature in directory on page 172
- -LicBorrow: Borrow license feature on page 173
- -LicWait: Wait until floating license is available from floating License Server on page 175

-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 on page 171
- -LicBorrow: Borrow license feature on page 173
- -LicWait: Wait until floating license is available from floating License Server on page 175

-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 on page 171
- -LicA: License information about every feature in directory on page 172
- -LicWait: Wait until floating license is available from floating License Server on page 175

-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 on page 171
- -LicA: License information about every feature in directory on page 172
- -LicBorrow: Borrow license feature on page 173

-M (-Ms, -Mb, -Ml): Memory Model

Group

Code Generation

Scope

Application

Syntax

`-M(s|b|l)`

Arguments

`s`: small memory model

`b`: banked memory model

`l`: large memory model.

Default

`-Ms`

Description

The Assembler for the MC68HC(S)12 supports three different memory models. The default is the small memory model, which corresponds to the normal setup, i.e., a 64kB code-address space. If you use some code memory expansion scheme, you may use banded memory model. The large memory model is used when using both a code and data memory expansion scheme.

Memory models are interesting when mixing ANSI-C and assembler files. For compatibility reasons, the different files must use the identical memory model.

Example

```
ASMOPTIONS=-Ms
```

-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. When the nesting level is reached, then the message A

Example

See the description of message A1004 for an example.

See also

Message A1004 (available in the online help)

-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 MCUasm Compatibility on page 433.

Example

```
ASMOPTIONS=-MCUasm
```

-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.

-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
```

-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
```

-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 on page 97

-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 on page 114

-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 executable name instead of "`assembler`".)

See also

Environment on page 97

-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 on page 361

-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 5.47 on page 187):

Listing 5.47 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
```

-View: Application standard occurrence

Group

Host

Scope

Assembly Unit

Syntax

`-View<kind>`

Arguments

`<kind>` is one of:

- `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 (e.g., Assembler, Linker, Compiler, ...) 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 on page 179 assembler option, a dialog box is still possible.

Example

```
C:\Freescale\prog\linker.exe -ViewHidden fibo.prm
```

-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
```

-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
```

-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 on page 223

-WOutFile: Create error listing file on page 222

-Wmsg8x3: Cut filenames in Microsoft format to 8.3

Group

Messages

Scope

Assembly Unit

Syntax

`-Wmsg8x3`

Arguments

None

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 Win95, WinNT, or a newer Windows O/S, 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

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFonp: Message format for no position information on page 211

-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.

-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.

-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.

-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.

-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.

-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 Listing 5.48 on page 201 is to be assembled in the batch mode.

Listing 5.48 Example assembly source code

```
var1:   equ  5
var2:   equ  5
  if (var1=var2)
    NOP
  endif
endif
```

The Assembler generates the error output (Listing 5.49 on page 201) in the assembler window if it is running in batch mode:

Listing 5.49 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 5.50 on page 201):

Listing 5.50 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 on page 109

Assembler options:

- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFonf: Message format for no file information on page 209

- -WmsgFonp: Message format for no position information on page 211

-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgFi[v|m]`

Arguments

`v`: Verbose format

`m`: Microsoft format

Default

`-WmsgFiv`

Description

If the Assembler is started without additional arguments (e.g., files to be assembled together with Assembler options), the Assembler is in the interactive mode (that is, a window is visible).

While in interactive mode, the Assembler uses the default verbose error file format to write the assembler messages (errors, warnings, information messages).

Using this option, the default format may be changed from verbose (with source, line and column information) to the *Microsoft* format (which displays only line information).

NOTE Using the Microsoft format may speed up the assembly process because the Assembler has to write less information to the screen.

Example

If the Assembler is running in interactive mode, the default error output is shown in the assembler window as in Listing 5.52 on page 204.

Listing 5.51 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 5.52 on page 204):

Listing 5.52 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 on page 109

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFonf: Message format for no file information on page 209
- -WmsgFonp: Message format for no position information on page 211

-WmsgFob: Message format for batch mode

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFob<string>
```

Arguments

`<string>`: format string (see Listing 5.53 on page 205)

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 Listing 5.53 on page 205 are supported (assumed that the source file is `x:\Freescale\sourcefile.asmx`).

Listing 5.53 Supported formats for messages in the batch node

```
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 Listing 5.54 on page 206 using the format. The options are set for producing the path of a file with its filename, extension, and line.

Listing 5.54 Message format in batch mode

```
x:\Freescale\sourcefile.asmx(3): error A1051: Right
 parenthesis expected
```

See also

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFonf: Message format for no file information on page 209
- -WmsgFonp: Message format for no position information on page 211

-WmsgFoi: Message format for interactive mode

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgFoi<string>`

Arguments

`<string>`: format string (see Listing 5.55 on page 207)

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 formats in Listing 5.55 on page 207 are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`):

Listing 5.55 Supported formats for the 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 format (Listing 5.56 on page 208):

Listing 5.56 Message format for interactive mode

```
x:\Freescale\sourcefile.asmx(3): error A1051: Right parenthesis
expected
```

See also

ERRORFILE: Filename specification error on page 109 environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFonf: Message format for no file information on page 209
- -WmsgFonp: Message format for no position information on page 211

-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 formats in Listing 5.57 on page 209 are supported:

Listing 5.57 Supported formats for the "no file information option"

```
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 on page 109 environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFonp: Message format for no position information on page 211

-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 formats in Listing 5.58 on page 211 are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`)

Listing 5.58 Supported formats for the “no position information” option

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

`\n`	`New line`

Example

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

produces a message in following format:

```
information L10324: Linking successful
```

See also

ERRORFILE: Filename specification error on page 109 environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 200
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 203
- -WmsgFob: Message format for batch mode on page 205
- -WmsgFoi: Message format for interactive mode on page 207
- -WmsgFonf: Message format for no file information on page 209

-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 on page 214
- -WmsgNw: Number of Warning messages on page 217

-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 on page 213
- -WmsgNw: Number of Warning messages on page 217

-WmsgNu: Disable user messages

Group

Messages

Scope

None

Syntax

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

Arguments

`a`: Disable messages about include files

`b`: Disable messages about reading files

`c`: Disable messages about generated files

`d`: Disable messages about processing statistics

`e`: Disable informal messages

Default

None

Description

The application produces some messages which are not in the normal message categories (WARNING, INFORMATION, ERROR, or FATAL). With this option such messages can be disabled. The purpose for this option is to reduce the amount of messages and to simplify the error parsing of other tools.

- `a`

  The application provides information about all included files. With this suboption this option can be disabled.

- `b`

  With this suboption messages about reading files e.g., the files used as input can be disabled.

- `c`

  Disables messages informing about generated files.

- `d`

  At the end of the assembly, the application may provide information about statistics, e.g., code size, RAM/ROM usage, and so on. With this suboption this option can be disabled.

- `e`

  With this option, informal messages (e.g., memory model, floating point format, ...) can be disabled.

NOTE Depending on the application, not all suboptions may make sense. In that case, they are just ignored for compatibility.

Example

```
-WmsgNu=c
```

-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 on page 213
- -WmsgNi: Number of Information messages on page 214

-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 on page 219
- -WmsgSi: Setting a message to Information on page 220
- -WmsgSw: Setting a Message to Warning on page 221

-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

Assembler options:

- -WmsgSd: Setting a message to disable on page 218
- -WmsgSi: Setting a message to Information on page 220
- -WmsgSw: Setting a Message to Warning on page 221

-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 on page 218
- -WmsgSe: Setting a message to Error on page 219
- -WmsgSw: Setting a Message to Warning on page 221

-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 on page 218
- -WmsgSe: Setting a message to Error on page 219
- -WmsgSi: Setting a message to Information on page 220

-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 on page 109.

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 on page 192
- -WStdout: Write to standard output on page 223

-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 is 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 on page 192
- -WOutFile: Create error listing file on page 222

6

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 on page 225 and Section types on page 226. The last part of the chapter deals with the merits of using relocatable sections. (See "Relocatable vs. absolute sections" on page 231.)

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.

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.

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.

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.

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 on page 226
- Relocatable sections on page 228

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 on page 330 assembler directive. The operand specified in the `ORG` directive determines the start address of the absolute section. See Listing 6.1 on page 226 for an example of constructing absolute sections using the `ORG` assembler directive.

Listing 6.1 Example source code using ORG for absolute sections

```
XDEF entry
      ORG   $A00  ; Absolute constant data section.
cst1: DC.B  $A6
cst2: DC.B  $BC
...
      ORG   $800  ; Absolute data section.
var:  DS.B  1

      ORG   $C00  ; Absolute code section.
```

```
entry:
      LDAA  cst1  ; Loads value in cst1
      ADDA  cst2  ; Adds value in cst2
      STAA  var   ; Stores into var
      BRA   entry
```

In the previous example, two bytes of storage are allocated starting at address `$A00`. The *constant* variable - `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.

The `PRM` file used to link the example above, can be defined as in Listing 6.2 on page 227.

Listing 6.2 Example PRM file for Listing 6.1 on page 226

```
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 0xFDFF;
/* 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 0xFFFE 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)

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 on page 338 assembler directive. See Listing 6.3 on page 228 for an example using the `SECTION` directive.

Listing 6.3 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:
          LDAA   cst1  ; Loads value into cst1
          ADDA   cst2  ; Adds value in cst2
          STAA   var   ; Stores 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.

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 in Listing 6.4 on page 229.

Listing 6.4 PRM file for Listing 6.3 on page 228 defining one RAM area and one ROM area

```
LINK test.abs /* Name of the executable file generated.      */
NAMES
  test.o      /* Name of the object file in the application. */
END
SECTIONS
/* READ_ONLY memory area.  */
  MY_ROM = READ_ONLY  0x0B00 TO 0x0BFF;
/* READ_WRITE memory area. */
  MY_RAM = READ_WRITE 0x0800 TO 0x08FF;
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 0xFFFE 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 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 `0x0800`.
- the `constSec` section will be allocated starting at `0x0B00`.
- the `codeSec` section will be allocated next to the `constSec` section.

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 in Listing 6.5 on page 230:

Listing 6.5 PRM file for Listing 6.3 on page 228 defining multiple RAM and ROM areas

```
LINK test.abs   /* Name of the executable file generated.      */
NAMES
  test.o        /* Name of the object file in the application. */
END
SECTIONS
 /* Two READ_ONLY memory areas */
  ROM_AREA_1= READ_ONLY  0x8000 TO 0x800F;
  ROM_AREA_2= READ_ONLY  0x8010 TO 0xFDFF;
/* 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;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec              INTO RAM_AREA_2;
  DEFAULT_RAM          INTO RAM_AREA_1;

/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec             INTO ROM_AREA_2;
  codeSec, DEFAULT_ROM INTO ROM_AREA_1;
END
INIT   entry                /* Application's entry point.          */
```

```
VECTOR 0 entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

- The name of the absolute file (`LINK` command).
- The name of the object files which should be linked (`NAMES` command).
- The specification of memory areas where the sections containing variables must be allocated. At least, the predefined `DEFAULT_RAM` section (or its ELF alias `'.data'`) must be placed there (`SECTIONS` and `PLACEMENT` commands).
- The specification of memory areas where the sections containing code or constants must be allocated. At least the predefined `DEFAULT_ROM` section (or its ELF alias `'.text'`) must be placed there (`SECTIONS` and `PLACEMENT` commands).
- The specification of the application entry point (`INIT` command)
- The definition of the reset vector (`VECTOR` command)

According to the PRM file in Listing 6.5 on page 230,

- 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`.

Relocatable vs. absolute sections

Generally, we recommend developing applications using relocatable sections. Relocatable sections offer several advantages.

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.

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.

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.

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.

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.

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.

7

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 a:

- Comment line on page 233 or a
- Source line on page 233.

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 (Listing 7.1 on page 233). 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 7.1 Examples of comments

```
; This is a comment line followed by an empty line and non comments

... (non comments)
```

Source line

Each source statement includes one or more of the following four fields:

- a Label field on page 234,
- an Operation field on page 234,
- 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 on page 133) is activated.

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 macro 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 on page 178 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`.

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 on page 234
- a Directive on page 246 name, or
- a Macro on page 246 name.

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).

Table 7.1 on page 235 presents an overview of the instructions available:

Table 7.1 HC12, HCS12, and HCS12X Instruction set

Instruction	Description
ABA	Add accumulator A and B
ABX	Add accumulator B and register X
ABY	Add accumulator B and register Y
ADCA	Add with Carry to accumulator A
ADCB	Add with Carry to accumulator B
ADDA	Add without carry to accumulator A
ADDB	Add without carry to accumulator B
ADDD	Add without carry to accumulator D
ADDX	Add without Carry to register X
ADDY	Add without Carry to register Y
ADDED	Add with Carry to accumulator D
ADEX	Add with Carry to register X
ADEY	Add with Carry to register Y
ANDA	Logical AND with accumulator A
ANDB	Logical AND with accumulator B
ANDCC	Logical AND with CCR
ANDX	Logical AND with register X
ANDY	Logical AND with register Y
ASL	Arithmetic Shift Left in memory
ASLA	Arithmetic Shift Left accumulator A
ASLB	Arithmetic Shift Left accumulator B

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
ASLD	Arithmetic Shift Left accumulator D
ASLW	Arithmetic Shift Left in memory
ASLX	Arithmetic Shift Left register X
ASLY	Arithmetic Shift Left register Y
ASR	Arithmetic Shift Right in memory
ASRA	Arithmetic Shift Right accumulator A
ASRB	Arithmetic Shift Right accumulator B
ASRW	Arithmetic Shift Right in memory
ASRX	Arithmetic Shift Right register X
ASRY	Arithmetic Shift Right register Y
BCC	Branch if Carry Clear
BCLR	Clear Bits in memory
BCS	Branch if Carry Set
BEQ	Branch if Equal
BGE	Branch if Greater than or Equal
BGND	Place in BGND mode
BGT	Branch if Greater Than
BHI	Branch if Higher
BHS	Branch if Higher or Same
BITA	Logical AND accumulator A and memory
BITB	Logical AND accumulator B and memory
BITX	Logical AND register X and memory
BITY	Logical AND register Y and memory
BLE	Branch if Less than or Equal
BLO	Branch if Lower (same as BCS)

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
BLS	Branch if Lower or Same
BLT	Branch if Less Than
BMI	Branch if Minus
BNE	Branch if Not Equal
BPL	Branch if Plus
BRA	Branch Always
BRCLR	Branch if bit Clear
BRN	Branch Never
BRSET	Branch if bits Set
BSET	Set Bits in memory
BSR	Branch to Subroutine
BTAS	Bit(s) Test and Set in memory
BVC	Branch if overflow Cleared
BVS	Branch if overflow Set
CALL	Call subroutine in extended memory
CBA	Compare accumulators A and B
CLC	Clear Carry bit
CLI	Clear Interrupt bit
CLR	Clear memory
CLRA	Clear accumulator A
CLRB	Clear accumulator B
CLRW	Clear memory
CLRX	Clear register X
CLRY	Clear register Y
CLV	Clear two’s complement overflow bit

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
CMPA	Compare memory with accumulator A
CMPB	Compare memory with accumulator B
COM	One’s complement on memory location
COMA	One’s complement on accumulator A
COMB	One’s complement on accumulator B
COMW	Complement memory
COMX	One’s complement on register X
COMY	One’s complement on register Y
CPD	Compare accumulator D and memory
CPED	Compare accumulator D and memory with borrow
CPES	Compare register SP and memory with borrow
CPEX	Compare register X and memory with borrow
CPEY	Compare register Y and memory with borrow
CPS	Compare register SP and memory
CPX	Compare register X and memory
CPY	Compare register Y and memory
DAA	Decimal Adjust Accumulator A
DBEQ	Decrement counter and Branch if zero
DBNE	Decrement counter and Branch if Not zero
DEC	Decrement memory location
DECA	Decrement accumulator A
DECB	Decrement accumulator B
DECW	Decrement memory location
DECX	Decrement register X
DECY	Decrement register Y

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
DES	Decrement register SP
DEX	Decrement index register X
DEY	Decrement index register Y
EDIV	Unsigned Division 32-bit/16-bit
EDIVS	Signed Division 32-bit/16-bit
EMACS	Multiply and Accumulate Signed
EMAXD	Get Maximum of two unsigned integers in accumulator D
EMAXM	Get Maximum of two unsigned integers in memory
EMIND	Get Minimum of two unsigned integers in accumulator D
EMINM	Get Minimum of two unsigned integers in Memory
EMUL	16-bit * 16-bit Multiplication (unsigned)
EMULS	16-bit * 16-bit Multiplication (Signed)
EORA	Logical XOR with accumulator A
EORB	Logical XOR with accumulator B
EORX	Logical XOR with register X
EORY	Logical XOR with register Y
ETBL	16-bit Table Lookup and Interpolate
EXG	Exchange register content
FDIV	16-bit /16-bit Fractional Divide
GLDAA	Load accumulator A from Global memory
GLDAB	Load accumulator B from Global memory
GLDD	Load accumulator D from Global memory
GLDS	Load register SP from Global memory
GLDX	Load register X from Global memory
GLDY	Load register Y from Global memory

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
GSTAA	Store accumulator A to Global memory
GSTAB	Store accumulator B to Global memory
GSTD	Store accumulator D to Global memory
GSTS	Store register SP to Global memory
GSTX	Store register X to Global memory
GSTY	Store register Y to Global memory
IBEQ	Increment counter and Branch if zero
IBNE	Increment counter and Branch if not zero
IDIV	16-bit /16-bit Integer Division (unsigned)
IDIVS	16-bit /16-bit Integer Division (Signed)
INC	Increment memory location
INCA	Increment accumulator A
INCB	Increment accumulator B
INCW	Increment memory location
INCX	Increment register X
INCY	Increment register Y
INS	Increment register SP
INX	Increment register X
INY	Increment register Y
JMP	Jump to label
JSR	Jump to Subroutine
LBCC	Long Branch if Carry Clear
LBCS	Long Branch if Carry Set
LBEQ	Long Branch if Equal
LBGE	Long Branch if Greater than or Equal

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
LBGT	Long Branch if Greater Than
LBHI	Long Branch if Higher
LBHS	Long Branch if Higher or Same
LBLE	Long Branch if Less than or Equal
LBLO	Long Branch if Lower (same as BCS)
LBLS	Long Branch if Lower or Same
LBLT	Long Branch if Less Than
LBMI	Long Branch if Minus
LBNE	Long Branch if Not Equal
LBPL	Long Branch if Plus
LBRA	Long Branch Always
LBRN	Long Branch Never
LBSR	Long Branch Subroutine
LBVC	Long Branch if overflow Clear
LBVS	Long Branch if overflow Set
LDAA	Load Accumulator A
LDAB	Load Accumulator B
LDD	Load accumulator D
LDS	Load register SP
LDX	Load index register X
LDY	Load index register Y
LEAS	Load SP with Effective Address
LEAX	Load X with Effective Address
LEAY	Load Y with Effective Address
LSL	Logical Shift Left in memory

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
LSLA	Logical Shift Left accumulator A
LSLB	Logical Shift Left accumulator B
LSLD	Logical Shift Left accumulator D
LSLW	Logical Shift Left in memory
LSLX	Logical Shift Left register X
LSLY	Logical Shift Left register Y
LSR	Logical Shift Right in memory
LSRA	Logical Shift Right Accumulator A
LSRB	Logical Shift right accumulator B
LSRD	Logical shift Right accumulator D
LSRW	Logical Shift Right in memory
LSRX	Logical Shift Right register X
LSRY	Logical Shift Right register Y
MAXA	Get Maximum of two unsigned bytes in accumulator A
MAXM	Get Maximum of two unsigned byte in Memory
MEM	Membership function
MOVW	Memory to memory word move
MINA	Get Minimum of two unsigned byte in accumulator A
MINM	Get Minimum of two unsigned byte in Memory
MOVB	Memory to memory Byte Move
MOVW	Memory to memory Word Move
MUL	8 * 8 bit unsigned Multiplication
NEG	2’s complement in memory
NEGA	2’s complement accumulator A
NEGB	2’s complement accumulator B

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
NEGW	2’s complement in memory
NEGX	2’s complement register X
NEGY	2’s complement register Y
NOP	No operation
ORAA	Logical OR with Accumulator A
ORAB	Logical OR with Accumulator B
ORCC	Logical OR with CCR
ORX	Logical OR register X with memory
ORY	Logical OR register Y with memory
PSHA	Push register A
PSHB	Push register B
PSHC	Push register CCR
PSHCW	Push register CCRW
PSHD	Push register D
PSHX	Push register X
PSHY	Push register Y
PULA	Pop register A
PULB	Pop register B
PULC	Pop register CCR
PULCW	Pop register CCRW
PULD	Pop register D
PULX	Pop register X
PULY	Pop register Y
REV	MIN-MAX Rule Evaluation for 8-bit values
REVW	MIN-MAX Rule Evaluation for 16-bit values

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
ROL	Rotate memory left
ROLA	Rotate accumulator A left
ROLB	Rotate accumulator B left
ROLW	Rotate memory left
ROLX	Rotate register X left
ROLY	Rotate register Y left
ROR	Rotate memory right
RORA	Rotate accumulator A Right
RORB	Rotate accumulator B Right
RORW	Rotate memory Right
RORX	Rotate register X Right
RORY	Rotate register Y Right
RTC	Return from CALL
RTI	Return from Interrupt
RTS	Return from Subroutine
SBA	Subtract accumulator A and B
SBCA	Subtract with Carry from accumulator A
SBCB	Subtract with Carry from accumulator B
SBED	Subtract with borrow from accumulator D
SBEX	Subtract with borrow from register X
SBEY	Subtract with borrow from register Y
SEC	Set carry bit
SEI	Set interrupt bit
SEV	Set two’s complement overflow bit
SEX	Sign Extend into 16-bit register

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
STAA	Store Accumulator A
STAB	Store Accumulator B
STD	Store Accumulator D
STOP	Stop
STS	Store register SP
STX	Store register X
STY	Store register Y
SUBA	Subtract without carry from accumulator A
SUBB	Subtract without carry from accumulator B
SUBD	Subtract without carry from accumulator D
SUBX	Subtract without carry from register X
SUBY	Subtract without carry from register Y
SWI	Software interrupt
TAB	Transfer A to B
TAP	Transfer A to CCR
TBA	Transfer B to A
TBEQ	Test counter and branch if zero
TBL	8-bit Table Lookup and Interpolate
TBNE	Test counter and branch if Not zero
TFR	Transfer Register to register
TPA	Transfer CCR to A
TRAP	Software Interrupt
TST	Test memory for 0 or minus
TSTA	Test accumulator A for 0 or minus
TSTB	Test accumulator B for 0 or minus

Table 7.1 HC12, HCS12, and HCS12X Instruction set (*continued*)

Instruction	Description
TSTW	Test memory for 0 or minus
TSTX	Test register X for 0 or minus
TSTY	Test register Y for 0 or minus
TSX	Transfer SP to X
TSY	Transfer SP to Y
TXS	Transfer X to SP
TYS	Transfer Y to SP
WAI	Wait for Interrupt
WAV	Weighted Average Calculation
XGDX	Exchange D with X
XGDY	Exchange D with Y

Directive

Assembler directives are described in the “Assembler Directives” on page 277 chapter of this manual.

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” on page 347 chapter in this manual.

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 addressing mode notations in Table 7.2 on page 247 are allowed in the operand field:

Table 7.2 HC(S)12 addressing modes

Addressing Mode	Notation
Inherent on page 248	No operands
Immediate on page 248	#<immediate 8-bit expression> or #<immediate 16-bit expression>
Direct on page 249	<8-bit address>
Extended on page 250	<16-bit address>
Relative on page 250	<PC relative, 8-bit offset> or <PC relative, 16-bit offset>
Indexed, 5-bit offset on page 251	<5-bit offset>, xysp
Indexed, 9-bit offset on page 252	<9-bit offset>, xysp
Indexed, 16-bit offset on page 253	<16-bit offset>, xysp
Indexed, indirect 16-bit offset on page 253	[<16-bit offset>, xysp]
Indexed, pre-decrement on page 254	<3-bit offset>, -xys
Indexed, pre-increment on page 255	<3-bit offset>, +xys
Indexed, post-decrement on page 255	<3-bit offset>, xys-
Indexed, post-increment on page 256	<3-bit offset>, xys+
Indexed, accumulator offset on page 257	abd, xysp
Indexed-indirect, accumulator D offset on page 257	[D, xysp]
Global on page 258	New instructions started with the label G are created for this usage like (GLDAA, GSTAA,...).

In the table above:

- xysp stands for one of the index registers `X`, `Y`, `SP`, `PC`, or `PCR`
- xys stands for one of the index registers `X`, `Y`, or `SP`
- abd stands for one of the accumulators `A`, `B`, or `D`

Inherent

Instructions using this addressing mode have no operands or all operands are stored in internal CPU registers (Listing 7.2 on page 248). The CPU does not need to perform any memory access to complete the instruction.

Listing 7.2 Inherent addressing-mode instructions

```
   NOP     ; Instruction with no operand
   CLRA    ; The operand is in the A CPU register
```

Immediate

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 (Listing 7.3 on page 248).

Listing 7.3 Immediate addressing mode

```
main:       LDAA   #$64
            LDX    #$AFE
            BRA    main
```

In this example, the hexadecimal value `$64` is loaded in the A register. The size of the immediate operand is implied by the instruction context. The A register is an 8-bit register, so the `LDAA` instruction expects an 8-bit immediate operand. The X register is a 16-bit register, so the `LDX` instruction expects a 16-bit immediate operand.

The immediate addressing mode can also be used to refer to the address of a symbol (Listing 7.4 on page 248).

Listing 7.4 Using the immediate addressing mode to refer to the address of a symbol

```
            ORG    $80
var1:       DC.B   $45, $67
            ORG    $800
main:
            LDX    #var1
            BRA    main
```

In this example, the address of the variable `'var1'` (`$80`) is loaded in the X register.

One very common programming error is to omit the # character. This causes the Assembler to misinterpret the expression as an address rather than an explicit data (Listing 7.5 on page 249).

Listing 7.5 Potential error - direct addressing mode instead of immediate

```
          LDAA  $60
```

means to load accumulator A with the value stored at address $60.

Direct

On the HC(S)12, the direct addressing mode is used to address operands in the direct page of the memory (location `$0000 to $00FF`, also called the zero page). On the HCS12X the direct page register (DIRECT) determines the position of the direct page within the memory map.The direct-addressing mode is used to access operands in the address range $00 through $FF in the direct page. Accesses on this memory range are faster and require less code than using the extended addressing mode (see Listing 7.6 on page 249). In order to speed up an application a programmer can decide to place the most commonly accessed data in this area of memory.

Listing 7.6 Direct addressing mode in an absolute section

```
           ORG $50
data:      DS.B 1

MyCode:    SECTION
Entry:
           LDS  #$AFE           ; init Stack Pointer
           LDAA #$01
main:      STAA data
           BRA main
```

In this example, the value in the A register is stored in the data variable which is located at address $50.

In Listing 7.7 on page 250, data1 is located in a relocatable section. To inform the Assembler that this section will be placed in the zero page, the SHORT qualifier after SECTION is used. The data2 label is imported into this code. To inform the Assembler that this label can also be used with the direct addressing mode, the "XREF.B" directive is used.

Listing 7.7 Direct addressing modes in a relocatable section

```
MyData:     SECTION SHORT
data1:      DS.B    1
            XREF.B data2
MyCode:     SECTION
Entry:
            LDS     #$AFE           ; init Stack Pointer
            LDAA    data1
main:       STAA    data2
            BRA     main
```

Extended

The extended addressing mode is used to access any memory location in the 64-Kilobyte memory map. In Listing 7.8 on page 250, the value in the `A` register is stored in the variable data. This variable is located at address `$0100` in the memory map.

Listing 7.8 Extended addressing mode

```
            XDEF   Entry
            ORG    $100
data:       DS.B   1
MyCode:     SECTION
Entry:
            LDS    #$AFE              ; init Stack Pointer
            LDAA #$01
main:       STAA data
            BRA main
```

Relative

This addressing mode is used to determine the destination address of branch instructions. Each conditional branch instruction tests some bits in the condition code register. If the bits are in the expected state, the specified offset is added to the address of the instruction following the branch instruction, and execution continues at that address.

Short branch instructions (`BRA`, `BEQ`, ...) expect a signed offset encoded on one byte. The valid range for a short branch offset is [`-128..127`]. In Listing 7.9 on page 251, after the two `NOP`s have been executed, the application branches on the first `NOP` and continues execution.

Listing 7.9 Relative addressing mode

```
main:
       NOP
       NOP
       BRA main
```

Long branch instructions (LBRA, LBEQ, ...) expect a signed offset encoded on two bytes. The valid range for a long branch offset is [-32768..32767].

Using the special symbol for the location counter, you can also specify an offset to the location pointer as the target for a branch instruction. The * refers to the beginning of the instruction where it is specified. In Listing 7.10 on page 251, after the two NOPs have been executed, the application branches at offset -2 from the BRA instruction (i.e., on the 'main' label).

Listing 7.10 Using BRA with an offset

```
main:
       NOP
       NOP
       BRA   *-2
```

Inside of an absolute section, expressions specified in a PC-relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in an XREF directive)
- an absolute EQU or SET label.

Inside of a relocatable section, expressions specified in a PC-relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in an XREF directive)

Indexed, 5-bit offset

This addressing mode add a 5-bit signed offset to the base index register to form the memory address that is referenced in the instruction. The valid range for a 5-bit signed offset is [-16..15]. The base index register may be X, Y, SP, PC, or PCR.

For information about the Indexed-PC and Indexed-PC-Relative addressing modes, see section 'Indexed PC vs. Indexed PC Relative Addressing Mode' below.

This addressing mode may be used to access elements in an n-element table, which size is smaller than 16 bytes (Listing 7.11 on page 252).

Listing 7.11 Indexed (5-bit offset) addressing mode

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
            ORG $800
DATA_TBL:   DS.B 10
main:
            LDX #CST_TBL
            LDAA 3,X

            LDY #DATA_TBL
            STAA 8, Y
```

The accumulator `A` is loaded with the byte value stored in memory location `$1003` `($1000 + 3)`.

Then the value of accumulator A is stored at address `$808` (`$800 + 8`).

Indexed, 9-bit offset

This addressing mode add a 9-bit signed offset to the base index register to form the memory address, which is referenced in the instruction. The valid range for a 9-bit signed offset is [`-256..255`]. The base index register may be `X`, `Y`, `SP`, PC, or `PCR`.

For information about Indexed-`PC` and Indexed-`PC`-Relative addressing modes, see the Indexed-PC vs. Indexed-PC relative addressing mode on page 258 section below.

This addressing mode may be used to access elements in an n-element table, whose size is smaller than 256 bytes (Listing 7.12 on page 252).

Listing 7.12 Indexed, 9-bit offset addressing mode

```
          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:
          LDX   #CST_TBL
          LDAA  20,X
```

```
LDY   #DATA_TBL
STAA  18, Y
```

Accumulator A is loaded with the byte value stored in memory location $1014 ($1000 + 20).

Then the value of accumulator A is stored at address $812 ($800 + 18).

Indexed, 16-bit offset

This addressing mode add a 16-bit offset to the base index register to form the memory address, which is referenced in the instruction. The 16-bit offset may be considered either as signed or unsigned ($FFFF may be considered to be -1 or 65,535). The base index register may be X, Y, SP, PC, or PCR.

For information about Indexed PC and Indexed PC Relative addressing mode, see the Indexed-PC vs. Indexed-PC relative addressing mode on page 258 section.

In Listing 7.13 on page 253, accumulator A is loaded with the byte value stored in memory location $900 ($600 + $300).

Then the value of accumulator A is stored at address $1140 ($1000 + $140).

Listing 7.13 Indexed, 16-bit offset addressing mode

```
main:
        LDX #$600
        LDAA $300,X

        LDY #$1000
        STAA $140, Y
```

Indexed, indirect 16-bit offset

This addressing mode adds a 16-bit offset to the base index register to form the address of a memory location containing a pointer to the memory location referenced in the instruction. The 16-bit offset may be considered either as signed or unsigned ($FFFF may be considered to be -1 or 65,535). The base index register may be X, Y, SP, PC, or PCR.

For information about Indexed-PC and Indexed-PC-Relative addressing mode, see section ‘Indexed PC vs. Indexed PC Relative Addressing Mode’.

In Listing 7.14 on page 254, the 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 accumulator A is loaded with $35, the value stored at address $2001.

Listing 7.14 Indexed, indirect 16-bit offset addressing mode

```
            ORG   $1000
CST_TBL1:   DC.W  $1020, $1050, $2001
            ORG   $2000
CST_TBL:    DC.B  $10, $35, $46
            ORG   $3000
main:
            LDX #CST_TBL1
            LDAA [4,X]
```

Indexed, pre-decrement

This addressing mode allow you to decrement the base register by a specified value, before indexing takes place. The base register is decremented by the specified value and the content of the modified base register is referenced in the instruction.

The valid range for a pre-decrement value is [1..8]. The base index register may be `X`, `Y`, or `SP`.

Listing 7.15 Indexed, pre-decrement addressing mode

```
            ORG   $1000
CST_TBL:    DC.B  $5, $10, $18, $20, $28, $30
END_TBL:    DC.B  $0
main:
            CLRA
            CLRB
            LDX   #END_TBL
loop:
            ADDD  1,-X
            CPX   #CST_TBL
            BNE   loop
```

In Listing 7.15 on page 254, the base register `X` is loaded with the address of the element following the table `CST_TBL ($1006)`.

The `X` register is decremented by 1 (its value is `$1005`) and the value at this address (`$30`) is added to register `D`.

X is not equal to the address of CST_TBL, so it is decremented again and the content of address ($1004) is added to D.

This loop is repeated as long as the X register did not reach the beginning of the table CST_TBL ($1000).

Indexed, pre-increment

This addressing mode allow you to increment the base register by a specified value, before indexing takes place. The base register is incremented by the specified value and the content of the modified base register is referenced in the instruction.

The valid range for a pre-increment value is [1..8]. The base index register may be X, Y, or SP.

In Listing 7.16 on page 255, the base register X is loaded with the address of the table CST_TBL ($1000). The register X is incremented by 2 (its value is $1002) and the value at this address ($18) is added to register D.

Listing 7.16 Indexed, pre-increment addressing mode

```
            ORG    $1000
CST_TBL:    DC.B   $5, $10, $18, $20, $28, $30
END_TBL:    DC.B   $0
main:
            CLRA
            CLRB
            LDX    #CST_TBL
loop:
            ADDD   2,+X
            CPX    #END_TBL
            BNE    loop
```

X is not equal to the address of END_TBL, so it is incremented again and the content of address ($1004) is added to D. This loop is repeated as long as the register X did not reach the end of the END_TBL ($1006) table.

Indexed, post-decrement

This addressing mode allow you to decrement the base register by a specified value, after indexing takes place. The content of the base register is read, and then the base register is decremented by the specified value.

The valid range for a pre-decrement value is [1..8]. The base index register may be X, Y, or SP.

In Listing 7.17 on page 256, the base register X is loaded with the address of the element following the table CST_TBL ($1006). The value at address $1006 ($0) is added to register D, and X is decremented by 2 (its value is $1004). X is not equal to the address of CST_TBL, so the value at address $1004 is added to D, and X is decremented by two again (its value is now $1002). This loop is repeated as long as the X register did not reach the beginning of the table CST_TBL ($1000).

Listing 7.17 Indexed, post-increment addressing mode

```
            ORG $1000
CST_TBL:    DC.B $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
            CLRA
            CLRB
            LDX   #END_TBL
loop:
            ADDD  2,X-
            CPX   #CST_TBL
            BNE   loop
```

Indexed, post-increment

This addressing mode allow you to increment the base register by a specified value, after indexing takes place. The content of the base register is read and then the base register is incremented by the specified value.

The valid range for a pre-increment value is [1..8]. The base index register may be X, Y, or SP.

In Listing 7.18 on page 256, the base register X is loaded with the address of the table CST_TBL ($1000). The value at address $1000 ($5) is added to register D and then the X register is incremented by 1 (its value is $1001). X is not equal to the address of END_TBL, so the value at address $1001 ($10) is added to register D and then the X register is incremented by 1 (its value is $1002). This loop is repeated as long as the X register did not reach the end of the table END_TBL ($1006).

Listing 7.18 Indexed, post-increment addressing mode

```
            ORG $1000

CST_TBL:    DC.B  $5, $10, $18, $20, $28, $30
END_TBL:    DC.B  $0
main:
            CLRA
```

```
            CLRB
            LDX #CST_TBL
loop:
            ADDD  1,X+
            CPX   #END_TBL
            BNE   loop
```

Indexed, accumulator offset

This addressing mode add the value in the specified accumulator to the base index register to form the address, which is referenced in the instruction. The base index register may be X, Y, SP, or PC. The accumulator may be A, B, or D.

In Listing 7.19 on page 257, the value stored in B ($20) is added to the value of X ($600) to form a memory address ($620). The value stored at $620 is loaded in accumulator A.

Listing 7.19 Indexed, accumulator offset addressing mode

```
main:
            LDAB  #$20
            LDX   #$600
            LDAA  B,X
            LDY   #$1000
            STAA  $140, Y
```

Indexed-indirect, accumulator D offset

This addressing mode add the value in D to the base index register to form the address of a memory location containing a pointer to the memory location referenced in the instruction. The base index register may be X, Y, SP or PC.

Listing 7.20 on page 257 is an example of jump table. The values beginning at goto1 are potential destination for the jump instruction. When JMP [D, PC] is executed, PC points to goto1 and D holds the value 2. The JMP instruction adds the value in D and PC to form the address of goto2. The CPU reads the address stored there (the address of the label entry2) and jumps there.

Listing 7.20 Index-indirect, accumulator D offset addressing mode

```
entry1:     NOP
            NOP
entry2:     NOP
            NOP
```

```
entry3:     NOP
            NOP
main:
            LDD    #2
            JMP    [D, PC]
goto1:      DC.W   entry1
goto2:      DC.W   entry2
goto3:      DC.W   entry3
```

Global

The physical address space on the HCS12 core architecture is limited to 64 KB. The HCS12X core architecture with the usage of the Global Page Index Register allows the accessing of up to 8 MB of memory. New instructions started with the label G are created for this usage.

In Listing 7.21 on page 258, Accumulator A is loaded from Global Memory. GLDAA has the same addressing mode like LDAA. However, the only difference is that memory address
(64 KB) is presented by the Global memory address (8 MB). This is the case for all Global instructions.

Listing 7.21 Global addressing mode

```
main:
            GLDAA $1020
            GSTAA $1020
```

Indexed-PC vs. Indexed-PC relative addressing mode

When using the indexed addressing mode with PC as the base register, the Macro Assembler allow you to use either Indexed-PC (<offset>, PC) or Indexed-PC Relative (<offset>, PCR) notation.

When Indexed-PC notation is used, the offset specified in inserted directly in the opcode (Listing 7.22 on page 258.

Listing 7.22 Using the indexed-PC addressing mode

```
main:
            LDAB   3, PC
```

```
            DC.B  $20, $30, $40, $50
```

In the example above, the register B is loaded with the value stored at address PC + 3 ($50).

When Indexed-PC-Relative notation is used, the offset between the current location counter and the specified expression is computed and inserted in the opcode.

In Listing 7.23 on page 259, the B register is loaded with the value at stored at label 'X4' ($50). The Macro Assembler evaluates the offset between the current location counter and the 'x4' symbol to determine the value, which must be stored in the opcode.

Listing 7.23 Using the indexed-PC relative addressing mode

```
main:
            LDAB  x4, PCR
x1:         DC.B  $20
x2:         DC.B  $30
x3:         DC.B  $40
x4:         DC.B  $50
```

Inside of an absolute section, expressions specified in an indexed PC-relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in an XREF directive)
- an absolute EQU or SET label.

Inside of a relocatable section, expressions specified in an indexed-PC relative addressing mode may be:

- a label defined in any absolute section
- a label defined in any relocatable section
- an external label (defined in an XREF directive)

Comment field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in the comment field. Listing 7.24 on page 260 shows a typical comment as the last field in a source statement.

Listing 7.24 Example of a comment

```
NOP ; Comment following an instruction
```

Symbols

The following types of symbols are the topics of this section:

- User-defined symbols on page 260
- External symbols on page 261
- Undefined symbols on page 261
- Reserved symbols on page 262

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 on page 338 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 (Listing 7.25 on page 260).

Listing 7.25 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 on page 340 or EQU - Equate symbol value on page 300 assembler directives.

Symbols with absolute values must be defined with constant expressions.

Listing 7.26 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.
```

External symbols

A symbol may be made external using the XDEF - External Symbol Definition on page 344 assembler directive. In another source file, an XREF - External Symbol Reference on page 345 assembler directive must reference it. Since its address is unknown in the referencing file, it is considered to be relocatable. See Listing 7.27 on page 261 for an example of using `XDEF` and `XREF`.

Listing 7.27 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
```

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. Listing 7.28 on page 261 shows an example of an undeclared label.

Listing 7.28 Example of an undeclared label

```
codeSec:  SECTION
entry:
     NOP
     BNE   entry
     NOP
     JMP   end
     JMP   label   ; <- Undeclared user-defined symbol: label
end: RTS
```

```
END
```

Reserved symbols

Reserved symbols cannot be used for user-defined symbols.

Register names are reserved identifiers.

For the HC12 processor these reserved identifiers are:

A, B, CCR, D, X, Y, SP, PC, PCR, TEMP1, TEMP2.

In addition, the keywords `HIGH`, `LOW` and `PAGE` are also a reserved identifier. It is used to refer to the bits 16-23 of a 24-bit value.

Constants

The Assembler supports integer and ASCII string constants:

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 on page 285 assembler directive. When the default base is not decimal, decimal values cannot be represented, because they do not have a prefix character.

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 Listing 7.29 on page 263 for a variety of string constants.

Listing 7.29 String constants

```
'ABCD', "ABCD", 'A', "'B", "A'B", 'A"B'
```

Floating-Point constants

The Macro Assembler does not support floating-point constants.

Operators

Operators recognized by the Assembler in expressions are:

- Addition and subtraction operators (binary) on page 263
- Multiplication, division and modulo operators (binary) on page 264
- Sign operators (unary) on page 265
- Shift operators (binary) on page 265
- Bitwise operators (binary) on page 266
- Logical operators (unary) on page 267
- Relational operators (binary) on page 268
- HIGH operator on page 269
- PAGE operator on page 270
- Force operator (unary) on page 270

Addition and subtraction operators (binary)

The addition and subtraction operators are + and –, respectively.

Syntax

```
Addition:     <operand> + <operand>
```

```
Subtraction: <operand> - <operand>
```

Description

The + operator adds two operands, whereas the − operator subtracts them. The operands can be any expression evaluating to an absolute or relocatable expression.

Addition between two relocatable operands is not allowed.

Example

See Listing 7.30 on page 264 for an example of addition and subtraction operators.

Listing 7.30 Addition and subtraction operators

```
$A3216 + $42 ; Addition of two absolute operands (= $A3258).
labelB - $10 ; Subtraction with value of 'labelB'
```

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 Listing 7.31 on page 265 for an example of the multiplication, division, and modulo operators.

Listing 7.31 Multiplication, division, and modulo operators

```
23 * 4     ; multiplication (= 92)
23 / 4     ; division (= 5)
23 % 4     ; remainder(= 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 Listing 7.32 on page 265 for an example of the unary sign operators.

Listing 7.32 Unary sign operators

```
+$32        ; ( = $32)
-$32        ; ( = $CE = -$32)
```

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 Listing 7.33 on page 266 for an example of the binary shift operators.

Listing 7.33 Binary shift operators

```
$25 << 2    ; shift left (= $94)
$A5 >> 3    ; shift right(= $14)
```

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 Listing 7.34 on page 267 for an example of the binary bitwise operators

Listing 7.34 Binary bitwise operators

```
$E & 3      ; = $2 (%1110 & %0011 = %0010)
$E | 3      ; = $F (%1110 | %0011 = %1111)
$E ^ 3      ; = $D (%1110 ^ %0011 = %1101)
```

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 Listing 7.35 on page 267 for an example of the unary bitwise operator.

Listing 7.35 Unary bitwise operator

```
~$C ; = $FFFFFFF3 (~%00000000 00000000 00000000 00001100
                   =%11111111 11111111 11111111 11110011)
```

Logical operators (unary)

The unary logical operator is `!`.

Syntax

```
Logical NOT: !<operand>
```

Description

The `!` operator returns 1 (true) if the operand is `0`, otherwise it returns `0` (false).

The operand can be any expression evaluating to an absolute expression.

Example

See Listing 7.36 on page 268 for an example of the unary logical operator.

Listing 7.36 Unary logical operator

```
!(8<5)     ; = $1 (TRUE)
```

Relational operators (binary)

The binary relational operators are =, ==, !=, <>, <, <=, >, and >=.

Listing 7.37 Syntax - relational operators

```
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 Listing 7.38 on page 269 for an example of the binary relational operators

Listing 7.38 Binary relational operators

```
3 >= 4       ; = 0  (FALSE)
label = 4    ; = 1  (TRUE) if label is 4, 0 or (FALSE) otherwise.
9 <  $B      ; = 1  (TRUE)
```

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.

```
LDAA  #HIGH(data1)
```

This instruction will load the immediate value of the high byte of the address of `data1` (`$10`) in register A.

```
LDAA  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.

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.

```
LDAA  #LOW(data1)
```

This instruction will load the immediate value of the lower byte of the address of `data1` (`$50`) in register A.

```
LDAA  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.

PAGE operator

The PAGE operator is `PAGE`.

Syntax

```
PAGE Byte: PAGE(<operand>)
```

Description

This operator returns the page byte of the address of a memory location.

Example

Assume `data1` is a word located at address `$28050` in the memory.

```
LDAA  #PAGE(data1)
```

This instruction will load the immediate value of the page byte of the address of `data1` (`$2`).

```
LDAA  PAGE(data1)
```

This instruction will load the direct value at memory location of the page byte of the address of `data1` (i.e., the value in memory location `$2`).

Force operator (unary)

The unary force operators are `<`, `.B`, `>`, and `.W`.

Syntax

8-bit address: `<<operand>` or `<operand>.B`

16-bit address: `><operand>` or `<operand>.W`

Description

The < or `.B` operators force the operand to be an 8-bit operand, whereas the > or `.W` operators force the operand to be a 16-bit operand.

`The` < operator may be useful to force the 8-bit immediate, 8-bit indexed, or direct addressing mode for an instruction.

> operator may be useful to force the 16-bit immediate, 16-bit indexed, or extended addressing mode for an instruction.

The operand can be any expression evaluating to an absolute or relocatable expression.

Example

```
<label          ; label is a 8-bit address.
label.B         ; label is a 8-bit address.
>label          ; label is a 16-bit address.
label.W         ; label is a 16-bit address.
```

Operator precedence

Operator precedence follows the rules for ANSI - C operators (Table 7.3 on page 271).

Table 7.3 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

Table 7.3 Operator precedence priorities (*continued*)

Operator	Description	Associativity
<< >>	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

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 on page 273: 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 on page 274: 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.

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 Listing 7.39 on page 273 for an example of an absolute expression.

Listing 7.39 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 7.40 Interpretation of label2-label1: difference between two relocatable symbols

```
(<offset label2> + <start section address >) -
(<offset label1> + <start section address >)
```

This can be simplified to (Listing 7.41 on page 273):

Listing 7.41 Simplified result for the difference between two relocatable symbols

```
<offset label2> + <start section address > -
<offset label1> - <start section address>
= <offset label2> - <offset label1>
```

Example

In the example in Listing 7.42 on page 274, 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 7.42 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
```

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 7.43 Example of relocatable expression

```
          XREF XtrnLabel
DataSec:  SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1
CodeSec:  SECTION
entry:
         LDAA tabBegin+2        ; Simple relocatable expression
         BRA   *-3              ; Simple relocatable expression
         LDAA XtrnLabel+6       ; Simple relocatable expression
```

Unary operation result

Table 7.4 on page 275 describes the type of an expression according to the operator in an unary operation:

Table 7.4 Expression type resulting from operator and operand type

Operator	Operand	Expression
-, !, ~	absolute	absolute
-, !, ~	relocatable	complex
+	absolute	absolute
+	relocatable	relocatable

Binary operations result

Table 7.5 on page 275 describes the type of an expression according to the left and right operators in a binary operation:

Table 7.5 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

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.
- Includes may be nested up to `50`.
- The maximum line length is `1023`.

8

Assembler Directives

There are different class of assembler directives. The following tables gives you an overview over the different directives and their class:

Directive overview

Section-Definition directives

The directives in Table 8.1 on page 277 are used to define new sections.

Table 8.1 Directives for defining sections

Directive	Description
ORG - Set Location Counter on page 330	Define an absolute section
SECTION - Declare Relocatable Section on page 338	Define a relocatable section
OFFSET - Create absolute symbols on page 328	Define an offset section

Constant-Definition directives

The directives in Table 8.2 on page 277 are used to define assembly constants.

Table 8.2 Directives for defining constants

Directive	Description
EQU - Equate symbol value on page 300	Assign a name to an expression (cannot be redefined)
SET - Set Symbol Value on page 340	Assign a name to an expression (can be redefined)

Data-Allocation directives

The directives in Table 8.3 on page 278 are used to allocate variables.

Table 8.3 Directives for allocating variables

Directive	Description
DC - Define Constant on page 288	Define a constant variable
DCB - Define Constant Block on page 290	Define a constant block
DS - Define Space on page 292	Define storage for a variable
RAD50 - Rad50-encoded string constants on page 335	RAD50 encoded string constants

Symbol-Linkage directives

Symbol-linkage directives (Table 8.4 on page 278) are used to export or import global symbols.

Table 8.4 Symbol linkage directives

Directive	Description
ABSENTRY - Application entry point on page 282	Specify the application entry point when an absolute file is generated
XDEF - External Symbol Definition on page 344	Make a symbol public (visible from outside)
XREF - External Symbol Reference on page 345	Import reference to an external symbol.
XREFB - External Reference for Symbols located on the Direct Page on page 346	Import reference to an external symbol located on the direct page.

Assembly-Control directives

Assembly-control directives (Table 8.5 on page 279) are general purpose directives used to control the assembly process.

Table 8.5 Assembly control directives

Directive	Description
ALIGN - Align Location Counter on page 284	Define Alignment Constraint
BASE - Set number base on page 285	Specify default base for constant definition
END - End assembly on page 296	End of assembly unit
ENDFOR - End of FOR block on page 297	End of FOR block
EVEN - Force word alignment on page 301	Define 2-byte alignment constraint
FAIL - Generate Error message on page 303	Generate user defined error or warning messages
FOR - Repeat assembly block on page 307	Repeat assembly blocks
INCLUDE - Include text from another file on page 313	Include text from another file.
LONGEVEN - Forcing Long-Word alignment on page 318	Define 4 Byte alignment constraint

Listing-File Control directives

Listing-file control directives (Table 8.6 on page 279) control the generation of the assembler listing file.

Table 8.6 Listing-file control directives

Directive	Description
CLIST - List conditional assembly on page 286	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable Listing on page 314	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set Line Length on page 316	Define line length in assembly listing file.

Table 8.6 Listing-file control directives (*continued*)

Directive	Description
MLIST - List macro expansions on page 322	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable Listing on page 325	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable Paging on page 327	Disable paging in the assembly listing file.
PAGE - Insert Page break on page 332	Insert page break.
PLEN - Set Page Length on page 334	Define page length in the assembler listing file.
SPC - Insert Blank Lines on page 341	Insert an empty line in the assembly listing file.
TABS - Set Tab Length on page 342	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide Listing Title on page 343	Define the user defined title for the assembler listing file.

Macro Control directives

Macro control directives (Table 8.7 on page 280) are used for the definition and expansion of macros.

Table 8.7 Macro control directives

Directive	Description
ENDM - End macro definition on page 299	End of user defined macro.
MACRO - Begin macro definition on page 319	Start of user defined macro.
MEXIT - Terminate Macro Expansion on page 320	Exit from macro expansion.

Conditional Assembly directives

Conditional assembly directives (Table 8.8 on page 281) are used for conditional assembling.

Table 8.8 Conditional assembly directives

Directive	Description
ELSE - Conditional assembly on page 294	alternate block
-Compat: Compatibility modes on page 137 assembler option on page 297	End of conditional block
IF - Conditional assembly on page 309	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly on page 311	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 or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined
IFNE	Test if an expression is not null.

Detailed descriptions of all assembler directives

The remainder of the chapter covers the detailed description of all available assembler directives.

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 Listing 8.1 on page 282 is assembled using the -FA2 assembler option, an ELF/DWARF 2.0 Absolute file is generated.

Listing 8.1 Using ABSENTRY to specify an application entry point

```
       ABSENTRY entry

       ORG   $fffe
Reset: DC.W  entry
       ORG   $70
entry: NOP
       NOP
main:  LDS #$1FFF
       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.

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 Listing 8.2 on page 284 aligns the `HEX` label to a location, which is a multiple of 16 (in this case, location `00010 (Hex)`)

Listing 8.2 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.
```

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 Listing 8.3 on page 285 for examples of setting the number base.

Listing 8.3 Setting the Number Base

```
    4    4                      base    10   ; default base: decimal
    5    5   000000 64          dc.b    100
    6    6                      base    16   ; default base: hex
    7    7   000001 0A          dc.b    0a
    8    8                      base    2    ; default base: binary
    9    9   000002 04          dc.b    100
   10   10   000003 04          dc.b    %100
   11   11                      base    @12  ; default base: decimal
   12   12   000004 64          dc.b    100
   13   13                      base    $a   ; default base: decimal
   14   14   000005 64          dc.b    100
   15   15
   16   16                      base    8    ; default base: octal
   17   17   000006 40          dc.b    100
```

Be careful. Even if the base value is set to 16, hexadecimal constants terminated by a `'D'` must be prefixed by the `$` character, otherwise they are supposed to be decimal constants in old style format. For example, constant `45D` is interpreted as decimal constant 45, not as hexadecimal constant `45D`.

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.

A soon as the -L: Generate a listing file on page 153 assembler option is activated, the Assembler defaults to CLIST ON.

Example

Listing 8.4 on page 286 is an example where the CLIST OFF option is used.

Listing 8.4 Listing file with CLIST OFF

```
      CLIST OFF
Try:  EQU     0
      IFEQ    Try
        LDAA  #103
      ELSE
        LDAA  #0
      ENDIF
```

Listing 8.5 on page 287 is the corresponding listing file.

Listing 8.5 Example assembler listing where CLIST OFF 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 8667                  LDAA  #103
    5    5                              ELSE
    7    7                              ENDIF
```

Listing 8.6 on page 287 is a listing file where `CLIST ON is used.`

Listing 8.6 CLIST ON is selected

```
       CLIST ON
Try:   EQU     0
       IFEQ    Try
         LDAA  #103
       ELSE
         LDAA #0
       ENDIF
```

Listing 8.7 on page 287 is the corresponding listing file.

Listing 8.7 Example assembler listing where CLIST ON is used

```
HC12-Assembler
 Abs. Rel.    Loc   Obj. code   Source line
 ---- ----   ------ ---------   -----------
    2    2          0000 0000   Try:    EQU   0
    3    3          0000 0000           IFEQ  Try
    4    4   000000 8667                  LDAA  #103
    5    5                              ELSE
    6    6                                LDAA #0
    7    7                              ENDIF
```

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 8.8 on page 288).
- `DC.W`: Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary (Listing 8.9 on page 288).
- `DC.L`: Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary (Listing 8.10 on page 289).

Listing 8.8 Example for DC.B

```
000000 4142 4344   Label: DC.B "ABCDE"
000004 45
000005 0A0A 010A          DC.B %1010, @12, 1,$A
```

Listing 8.9 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
```

Listing 8.10 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 value is truncated and a warning message is generated.

See also

Assembler directives:

- DCB - Define Constant Block on page 290
- DS - Define Space on page 292
- ORG - Set Location Counter on page 330
- SECTION - Declare Relocatable Section on page 338

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 the product: `<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 (Listing 8.11 on page 290):

- `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 8.11 Assembly output listing showing the allocation of constants

```
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 on page 288
- DS - Define Space on page 292
- ORG - Set Location Counter on page 330

- SECTION - Declare Relocatable Section on page 338

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 (Listing 8.12 on page 292). The content of the memory reserved is not initialized. The length of the block is the product:
`<size> * <count>`.

`<count>` may not contain undefined, forward, or external references. It may range from 1 to 4096.

Listing 8.12 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 Listing 8.13 on page 293 on page 293, 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. In order to allocate them separately, put them in different sections (Listing 8.14 on page 293).

Listing 8.13 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
```

Listing 8.14 How it should be done.

```
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

Assembler directives:

- DC - Define Constant on page 288
- ORG - Set Location Counter on page 330
- SECTION - Declare Relocatable Section on page 338

ELSE - Conditional assembly

Syntax

```
IF <condition>
  [<Block 1 - assembly language statements>]
[ELSE]
  [<Block 2 - 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

Listing 8.15 on page 294 is an example of the use of conditional assembly directives:

Listing 8.15 Various conditional assembly directives

```
Try: EQU     1
     IF Try  != 0
       LDAA  #103
     ELSE
       LDAA  #0
     ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the "`ldaa #103`" instruction is assembled. Changing the operand of the "`EQU`" directive to `0` causes the "`ldaa #0`" instruction to be assembled instead. Listing 8.16 on page 295 shows the listing provided by the Assembler for these lines of code:

Listing 8.16 Output listing of Listing 8.15 on page 294

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1            0000 0001   Try: EQU   1
   2    2            0000 0001        IF Try != 0
   3    3     000000 8667               LDAA  #103
   4    4                             ELSE
   6    6                             ENDIF
```

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 Listing 8.17 on page 296 causes any source code after the END statement to be ignored, as in Listing 8.18 on page 296.

Listing 8.17 Source File

```
Label:  DC.W  $1234
        DC.W  $5678
        END
        DC.W  $90AB ; no code generated
        DC.W  $CDEF ; no code generated
```

Listing 8.18 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
```

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 8.28 on page 307 in the `FOR`.section.

See also

FOR - Repeat assembly block on page 307 assembler directive

-Compat: Compatibility modes on page 137 assembler option

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 8.30 on page 309 in the `IF` section.

See also

IF - Conditional assembly on page 309 assembler directive

ENDM - End macro definition

Syntax

`ENDM`

Synonym

None

Description

The `ENDM` directive terminates the macro definition (Listing 8.19 on page 299).

Example

The `ENDM` statement in Listing 8.19 on page 299 terminates the `cpChar` macro.

Listing 8.19 Using ENDM to terminate a macro definition

```
cpChar:   MACRO
            LDAA   \1
            STAA   \2
          ENDM
DataSec:  SECTION
char1:    DS     1
char2:    DS     1
CodeSec:  SECTION
Start:
          cpChar char1, char2
```

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 Listing 8.20 on page 300 for examples of using the `EQU` directive.

Listing 8.20 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
```

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 Listing 8.21 on page 301 for instances where the EVEN directive causes padding bytes to be inserted.

Listing 8.21 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 on page 284 assembly directive

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-$FFFFFFFF]`, 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 Listing 8.22 on page 303 generates the error messages in Listing 8.23 on page 304. 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 8.22 Example source code

```
cpChar: MACRO
         IFC "\1", ""
           FAIL 200
            MEXIT
          ELSE
            LDAA  \1
```

```
          ENDIF

          IFC "\2", ""
            FAIL  600
          ELSE
            STAA  \2
          ENDIF
        ENDM
codSec: SECTION
Start:
        cpChar char1
```

Listing 8.23 Error messages resulting from assembling the source code in Listing 8.22 on page 303

```
>> in "C:\Freescale\demo\warnfail.asm", line 13, col 19, pos 226

          IFC "\2", ""
            FAIL  600
                   ^
WARNING A2332: FAIL found
Macro Call :            FAIL  600
```

Listing 8.24 on page 304 is another assembly code example which again incorporates the ‘`FAIL  200`’ and the ‘`FAIL  600`’ directives. Listing 8.25 on page 305 is the error message that was generated as a result of assembling the source code in Listing 8.24 on page 304.

Listing 8.24 Example source code

```
cpChar: MACRO
          IFC "\1", ""
            FAIL  200
            MEXIT
          ELSE
            LDAA  \1
          ENDIF

          IFC "\2", ""
            FAIL  600
          ELSE
            STAA  \2
          ENDIF
        ENDM
```

```
codeSec: SECTION
Start:
         cpChar, char2
```

Listing 8.25 Error messages resulting from assembling the source code in Listing 8.24 on page 304

```
>> in "C:\Freescale\demo\errfail.asm", line 6, col 19, pos 96

          IFC "\1", ""
            FAIL  200
                  ^
ERROR A2329: FAIL found
Macro Call :                 FAIL  200
```

Listing 8.26 on page 305 has additional uses of the FAIL directive. In this example, the ‘`FAIL  string`’ and ‘`FAIL  600`’ directives are used. Any error messages generated from the assembly code as a result of the FAIL directive are listed in Listing 8.27 on page 306.

Listing 8.26 Example source code

```
cpChar: MACRO
          IFC "\1", ""
            FAIL  "A character must be specified as first parameter"
            MEXIT
          ELSE
            LDAA  \1
          ENDIF

          IFC "\2", ""
            FAIL  600
          ELSE
            STAA  \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
         cpChar, char2
```

Listing 8.27 Error messages resulting from assembling the source code in **Listing 8.26 on page 305**

```
>> in "C:\Freescale\demo\failmes.asm", line 7, col 17, pos 110

          IFC "\1", ""

            FAIL  "A character must be specified as first parameter"
                  ^
ERROR A2338: A character must be specified as first parameter
Macro Call :   FAIL "A character must be specified as first parameter"
```

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

Listing 8.28 on page 307 is an example of using `FOR` to create a 5-repetition loop.

Listing 8.28 Using the FOR directive in a loop

```
FOR label=2 TO 6
   DC.B  label*7
ENDFOR
```

Listing 8.29 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

on page 297ENDFOR - End of FOR block on page 297

-Compat: Compatibility modes on page 137 assembler option

IF - Conditional assembly

Syntax

```
IF <condition>
 [<Block 1 - assembly language statements>]
[ELSE]
 [<Block 2 - 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

Listing 8.30 on page 309 is an example of the use of conditional assembly directives

Listing 8.30 IF and ENDIF

```
Try: EQU   0
     IF Try != 0
       LDAA  #103
     ELSE
       LDAA  #0
```

```
    ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the "`ldaa   #0`" instruction is assembled. Changing the operand of the "`EQU`" directive to one causes the "`ldaa   #103`" instruction to be assembled instead. The following shows the listing provided by the Assembler for these lines of code:

Listing 8.31 Output listing after conditional assembly

```
    1    1         0000 0000   Try: EQU   0
    2    2         0000 0000        IF Try != 0
    4    4                          ELSE
    4    4   000000 8667              LDAA  #103
    6    6                          ENDIF
```

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.

Table 8.9 on page 311 lists the available conditional types:

Table 8.9 Conditional assembly types

Ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>

Table 8.9 Conditional assembly types (*continued*)

Ifcc	Condition	Meaning
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Example

In Listing 8.32 on page 312 the value of `Try` determines the instruction to be assembled in the program. As shown, the "`ldaa #0`" instruction is assembled. Changing the directive to "`IFEQ`" causes the "`ldaa #103`" instruction to be assembled instead.

Listing 8.32 on page 312 is an example of the use of conditional assembler directives:

Listing 8.32 Using the IFNE conditional assembler directive

```
Try: EQU   0
     IFNE Try
       LDAA  #103
     ELSE
       LDAA  #0
     ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the "`ldaa #0`" instruction is assembled. Changing the directive to "`IFEQ`" causes the "`ldaa #103`" instruction to be assembled instead.

Listing 8.33 on page 312 shows the listing provided by the Assembler for these lines of code

Listing 8.33 output listing for Listing 8.32 on page 312

```
    1    1           0000 0000    Try: EQU   0
    2    2           0000 0000         IFNE Try
    4    4                             ELSE
    5    5    000000 8600                LDAA  #0
    6    6                             ENDIF
```

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 on page 112 environment variable.

Example

```
INCLUDE "..\LIBRARY\macros.inc"
```

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 on page 153 assembler option is specified on the command line.

The source text following the `LIST` directive is listed until a NOLIST - Disable Listing on page 325 or an END - End assembly on page 296 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 Listing 8.34 on page 314 generates the output listing in Listing 8.35 on page 315.

Listing 8.34 Using the LIST and NOLIST assembler directives

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP         NOP
```

Listing 8.35 Output listing generated from running Listing 8.34 on page 314

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   000000 A7          aaa:    NOP
   2    2
   4    4   000001 A7          bbb:    NOP
   5    5   000002 A7                  NOP
   6    6
  12   12   000005 A7          ddd:    NOP
  13   13   000006 A7                  NOP
```

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 portion of code in Listing 8.36 on page 316 generates the listing file in Listing 8.37 on page 316. Notice that the '`LLEN 24`' directive causes the output at the location-counter line 7 to be truncated.

Listing 8.36 Example assembly source code using LLEN

```
        DC.B  $55
        LLEN  32
        DC.W  $1234, $4567

        LLEN  24
        DC.W  $1234, $4567
        EVEN
```

Listing 8.37 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
```

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 Listing 8.38 on page 318 for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

Listing 8.38 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
```

See Also

ALIGN - Align Location Counter on page 284 assembler directive

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 Listing 8.39 on page 319 for a macro definition.

Listing 8.39 Example macro definition

```
           XDEF  Start
MyData:   SECTION
char1:    DS.B  1
char2:    DS.B  1
cpChar:   MACRO
            LDAA  \1
            STAA  \2
          ENDM
CodeSec:  SECTION
Start:
          cpChar char1, char2
```

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 on page 299 directive.

Example

See Listing 8.40 on page 320 allows the replication of simple instructions or directives using MACRO with MEXIT.

Listing 8.40 Example assembly code using MEXIT

```
         XDEF  entry
storage: EQU   $00FF

save:    MACRO              ; Start macro definition
           LDX   #storage
           LDAA  \1
           STAA  0,x        ; Save first argument
           LDAA  \2
           STAA  2,x        ; Save second argument
           IFC   '\3', ''   ; Is there a third argument?
             MEXIT          ; No, exit from macro.
             ENDC
           LDAA  \3         ; Save third argument
           STAA  4,X
         ENDM               ; End of macro definition

datSec:  SECTION
char1:   ds.b  1
char2:   ds.b  1
```

```
codSec: SECTION
entry:
        save  char1, char2
```

Listing 8.41 on page 321 shows the macro expansion of the previous macro.

Listing 8.41 Macro expansion of Listing 8.40 on page 320

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF  entry
    2    2
    3    3          0000 00FF   storage: EQU   $00FF
    4    4
    5    5                      save:    MACRO ; Start macro definiti
    6    6                                 LDX   #storage
    7    7                                 LDAA  \1
    8    8                                 STAA  0,x ; Save first arg
    9    9                                 LDAA  \2
   10   10                                 STAA  2,x ; Save second ar
   11   11                                 IFC '\3', '' ; Is there a
   12   12                                   MEXIT    ; No, exit macro
   13   13                                 ENDC
   14   14                                 LDAA  \3    ; Save third ar
   15   15                                 STAA  4,X
   16   16                               ENDM          ; End of macro
   17   17
   18   18                      datSec:  SECTION
   19   19   000000             char1:   ds.b  1
   20   20   000001             char2:   ds.b  1
   21   21
   22   22                      codSec:  SECTION
   23   23                      entry:
   24   24                               save  char1, char2
   25    6m   000000 CE 00FF    +        LDX   #storage
   26    7m   000003 B6 xxxx    +        LDAA  char1
   27    8m   000006 6A00       +        STAA  0,x    ; save first arg
   28    9m   000008 B6 xxxx    +        LDAA  char2
   29   10m  00000B 6A02        +        STAA  2,x    ; save second ar
   30   11m         0000 0001   +        IFC '', ''   ; Is there a 3rd
   32   12m                     +          MEXIT      ; No, exit macro
   33   13m                     +        ENDC
   34   14m                     +        LDAA         ; Save third arg
   35   15m                     +        STAA  4,X
```

MLIST - List macro expansions

Syntax

```
MLIST [ON|OFF]
```

Description

When the `ON` keyword is entered with an `MLIST` directive, the Assembler includes the macro expansions in the listing and in the debug file.

When the `OFF` keyword is entered, the macro expansions are omitted from the listing and from the debug file.

This directive is not written to the listing and debug file, and the default value is `ON`.

Synonym

None

Example

The assembly code in Listing 8.42 on page 322, with `MLIST ON`, generates the assembler output listing in Listing 8.43 on page 323

Listing 8.42 Example assembly source code

```
        XDEF   entry
        MLIST  ON
swap:   MACRO
          LDD    \1
          LDX    \2
          STD    \2
          STX    \1
        ENDM
codSec: SECTION
entry:
        LDD   #$F0
        LDX   #$0F
main:
        STD   first
        STX   second
        swap  first, second
        NOP
        BRA   main
```

```
datSec: SECTION
first:  DS.W   1
second: DS.W   1
```

Listing 8.43 Assembler output listing the example in Listing 8.42 on page 322 with MLIST ON

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                              XDEF  entry
    3    3                      swap:   MACRO
    4    4                                LDD   \1
    5    5                                LDX   \2
    6    6                                STD   \2
    7    7                                STX   \1
    8    8                              ENDM
    9    9                      codSec: SECTION
   10   10                      entry:
   11   11   000000 CC 00F0             LDD   #$F0
   12   12   000003 CE 000F             LDX   #$0F
   13   13                      main:
   14   14   000006 7C xxxx             STD   first
   15   15   000009 7E xxxx             STX   second
   16   16                              swap  first, second
   17    4m  00000C FC xxxx    +          LDD   first
   18    5m  00000F FE xxxx    +          LDX   second
   19    6m  000012 7C xxxx    +          STD   second
   20    7m  000015 7E xxxx    +          STX   first
   21   17   000018 A7                  NOP
   22   18   000019 20EB                BRA   main
   23   19                      datSec: SECTION
   24   20   000000             first:  DS.W  1
   25   21   000002             second: DS.W  1
```

For the same code, with `MLIST OFF`, the Assembler produces the listing file shown in Listing 8.44 on page 323.

Listing 8.44 Listing File with MLIST OFF

```
HC12-Assembler
```

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                              XDEF  entry
    3    3                      swap:   MACRO
    4    4                                LDD   \1
    5    5                                LDX   \2
    6    6                                STD   \2
    7    7                                STX   \1
    8    8                              ENDM
    9    9                      codSec: SECTION
   10   10                      entry:
   11   11   000000 CC 00F0             LDD   #$F0
   12   12   000003 CE 000F             LDX   #$0F
   13   13                      main:
   14   14   000006 7C xxxx             STD   first
   15   15   000009 7E xxxx             STX   second
   16   16                              swap  first, second
   21   17   000018 A7                  NOP
   22   18   000019 20EB                BRA   main
   23   19                      datSec: SECTION
   24   20   000000                     first:  DS.W  1
   25   21   000002                     second: DS.W  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.

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 on page 314 assembler directive is reached.

Example

See Listing 8.45 on page 325 for an example of using `LIST` and `NOLIST`.

Listing 8.45 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 Listing 8.46 on page 325.

Listing 8.46 Assembler output listing from the assembler source code in Listing 8.45 on page 325

```
HC12-Assembler
```

```
Abs. Rel.    Loc     Obj. code    Source line
---- ----    ------ ---------    -----------
   1    1    000000 A7           aaa:     NOP
   2    2
   4    4    000001 A7           bbb:     NOP
   5    5    000002 A7                    NOP
   6    6
  12   12    000005 A7           ddd:     NOP
  13   13    000006 A7                    NOP
```

See Also

LIST - Enable Listing on page 314 assembler directive

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.

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.

An offset section is useful to simulate data structures or a stack frame.

Examples

The example shown in Listing 8.47 on page 328 shows you how to use the `OFFSET` directive to access elements of a structure.

Listing 8.47 Using the OFFSET Directive

```
         OFFSET 0
ID:      DS.B   1
COUNT:   DS.W   1
VALUE:   DS.L   1
SIZE:    EQU    *

DataSec: SECTION
Struct:  DS.B   SIZE

CodeSec: SECTION
entry:
         LDX    #Struct
         LDAA   #0
         STAA   ID, X
         INC    COUNT, X
         INCA
         STAA   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 terminated. The preceding section is reactivated, and the location counter is restored to the next available location in this section.

See Listing 8.48 on page 329 for an example.

Listing 8.48 Example—Using the OFFSET Directive

```
 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 CExx xx              LDX   #Struct
   13   13   000003 8600                 LDAA  #0
   14   14   000005 6A00                 STAA  ID, X
   15   15   000007 6201                 INC   COUNT, X
   16   16   000009 42                   INCA
   17   17   00000A 6A03                 STAA  VALUE, X
```

In the example above, the `'cst3'` symbol, defined after the `OFFSET` directive, defines a constant byte value. This symbol is appended to the `'ConstSec'` section, which precedes the `OFFSET` directive.

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 on page 225 chapter).

Example

See Listing 8.49 on page 330 for an example where `ORG` sets the location counter.

Listing 8.49 Using ORG to set the location counter

```
      org   $2000
b1:   nop
b2:   rts
```

Viewing Listing 8.50 on page 330, you can see that the `b1` label is located at address `$2000` and label `b2` is at address `$2001`.

Listing 8.50 Assembler output listing from the source code in Listing 8.49 on page 330

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----  ------ ---------   -----------
    1    1                           org   $2000
    2    2  a002000 A7          b1:    nop
    3    3  a002001 3D          b2:    rts
```

See also

Assembler directives:

- DC - Define Constant on page 288
- DCB - Define Constant Block on page 290
- DS - Define Space on page 292
- SECTION - Declare Relocatable Section on page 338

PAGE - Insert Page break

Syntax

```
PAGE
```

Synonym

None

Description

Insert a page break in the assembly listing.

Example

The portion of code in Listing 8.51 on page 332 demonstrates the use of a page break in the assembler output listing.

Listing 8.51 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 Listing 8.52 on page 332.

Listing 8.52 Assembler output listing from the source code in Listing 8.51 on page 332

```
 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
```

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.

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 (Listing 8.53 on page 335).

Listing 8.53 RAD50 encoding

```
unsigned short LookUpPos(char x) {
  static const char translate[]=

    " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
  const char* pos= strchr(translate, x);
  if (pos == NULL) { EncodingError(); return 0; }
  return pos-translate;
}
unsigned short Encode(char a, char b, char c) {
  return LookUpPos(a)*40*40 + LookUpPos(b)*40
                             + LookUpPos(c);
}
```

If the remaining string is shorter than 3 bytes, it is filled with spaces (which correspond to the `RAD50` character 0).

The optional argument `cnt` can be used to explicitly state how many 16-bit values should be written. If the string is shorter than `3*cnt`, then it is filled with spaces.

See the example C code below (Listing 8.56 on page 336) about how to decode it.

Example

The string data in Listing 8.54 on page 336 assembles to the following data (Listing 8.55 on page 336). The 11 characters in the string are represented by 8 bytes.

Listing 8.54 RAD50 Example

```
  XDEF rad50, rad50Len
DataSection   SECTION
rad50:        RAD50 "Hello World"
rad50Len:     EQU (*-rad50)/2
```

Listing 8.55 Assembler output where 11 characters are contained in eight bytes

```
$32D4 $4D58 $922A $4BA0
```

This C code shown in Listing 8.56 on page 336 takes the data and prints “Hello World”.

Listing 8.56 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);
  }
```

```
}
```

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 Listing 8.57 on page 338 demonstrates the definition of a section `aaa`, which is split in 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 8.57 Example of the SECTION assembler directive

```
  1    1                          aaa:    SECTION 4
  2    2    000000 A7             xx:     NOP
  3    3                          bbb:    SECTION 5
  4    4    000000 A7             yy:     NOP
  5    5    000001 A7                     NOP
  6    6    000002 A7                     NOP
  7    7                          aaa:    SECTION 4
```

```
    8      8    000001 A7              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.

Example

The following example demonstrates the definition and usage of a SHORT section.

In the example shown in Listing 8.58 on page 339, the symbol data is accessed using the direct addressing mode.

Listing 8.58 Using the direct addressing mode

```
HC12-Assembler
 Abs. Rel.    Loc    Obj. code    Source line
 ---- ----    ------ ---------    -----------
    1     1                       dataSec: SECTION SHORT
    2     2   000000              data:     DS.B  1
    3     3
    4     4                       codeSec: SECTION
    5     5
    6     6                       entry:
    7     7   000000 87                     CLRA
    8     8   000001 5Axx                   STAA  data
```

See also

Assembler directives:

- ORG - Set Location Counter on page 330
- DC - Define Constant on page 288
- DCB - Define Constant Block on page 290
- DS - Define Space on page 292

SET - Set Symbol Value

Syntax

```
<label>: SET <expression>
```

Synonym

None

Description

Similar to the EQU - Equate symbol value on page 300 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 Listing 8.59 on page 340 for examples of the `SET` directive.

Listing 8.59 Using the SET assembler directive

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1          0000 0002   count:  SET   2
    2    2   000000 02          one:    DC.B  count
    3    3
    4    4          0000 0001   count:  SET   count-1
    5    5   000001 01                  DC.B  count
    6    6
    7    7          0000 0001           IFNE  count
    8    8          0000 0000   count:  SET   count-1
    9    9                              ENDIF
   10   10   000002 00                  DC.B  count
```

The value associated with the label `count` is decremented after each `DC.B` instruction.

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.

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.

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.

XDEF - External Symbol Definition

Syntax

`XDEF [.<size>] <label>[,<label>]...`

where `<size> = B(direct), W (default),` or `L`

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.

Example

See Listing 8.60 on page 344 for the case where the `XDEF` assembler directive can specify symbols that can be used by other modules.

Listing 8.60 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
```

XREF - External Symbol Reference

Syntax

```
XREF [.<size>] <symbol>[,<symbol>]...
```

where `<size> = B(direct)`, `W` (default), or `L`.

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.

Example

```
XREF OtherGlobal ; Reference "OtherGlobal" defined in
                 ; another module. (See the XDEF
                 ; directive example.)
```

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 an `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 an `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.)
```

9

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.

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.

Defining a macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.
- 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 on page 277 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.

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.

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 backlashes 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 macro definition in Listing 9.1 on page 348:

Listing 9.1 Example 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.

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, brackets, question marks and backward slashes can also be escaped with a backward slash as a 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 `([` or `])`, a question mark `(?)`, or a second backslash `(\)`.

Listing 9.2 Example macro definition

```
MyMacro:  MACRO
            DC     \1
          ENDM
MyMacro1: MACRO
            \1
          ENDM
```

Listing 9.3 on page 349 has some macro calls with rather complicated arguments:

Listing 9.3 Macro calls for Listing 9.2 on page 349

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro  [?$10, $56?]?]
```

```
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines (Listing 9.4 on page 350):

Listing 9.4 Macro expansion of Listing 9.3 on page 349

```
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 (Listing 9.5 on page 350):

Listing 9.5 Angle bracket syntax

```
      MyMacro <$10, $56>
```

CAUTION However, this old syntax is ambiguous, as < and > are also used as compare operators. For example, the following code (Listing 9.6 on page 350) does not produce the expected result:

Listing 9.6 Potential problem using the angle-bracket syntax

```
      MyMacro <1 > 2, 2 > 3> ; Wrong!
```

TIP 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 following assembler options:

- -CMacBrackets: Square brackets for macro arguments grouping on page 136
- -CMacAngBrack: Angle brackets for grouping Macro Arguments on page 135

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.

This is the definition of the clear macro (Listing 9.7 on page 351).

Listing 9.7 Clear macro definition

```
clear:   MACRO
           LDX    #\1
           LDAA   #16
\@LOOP:    CLR    1,X+
           DBNE   A,\@LOOP
         ENDM
```

This macro is called in the application (Listing 9.8 on page 351).

Listing 9.8 Calling the clear macro

```
Data: Section
temporary: DS  16
data:      DS  16

Code: Section
         clear temporary
         clear data
```

The two macro calls of `clear` are expanded in the manner shown in Listing 9.9 on page 351.

Listing 9.9 Example—Labels within Macros

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      clear:   MACRO
    2    2                                 LDX    #\1
    3    3                                 LDAA   #16
    4    4                      \@LOOP:    CLR    1,X+
    5    5                                 DBNE   A,\@LOOP
    6    6                               ENDM
```

```
  7    7
  8    8                           Data: Section
  9    9   000000                  temporary: DS    16
 10   10   000010                  data:      DS    16
 11   11
 12   12                           Code: Section
 13   13                                    clear temporary
 14    2m  000000 CE xxxx    +              LDX   #temporary
 15    3m  000003 8610       +              LDAA  #16
 16    4m  000005 6930       +_00001LOOP: CLR   1,X+
 17    5m  000007 0430 FB    +              DBNE  A,_00001LOOP
 18   14                                    clear data
 19    2m  00000A CE xxxx    +              LDX   #data
 20    3m  00000D 8610       +              LDAA  #16
 21    4m  00000F 6930       +_00002LOOP: CLR   1,X+
 22    5m  000011 0430 FB    +                 DBNE  A,_00002LOOP
```

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.

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 call are also supported.

A macro call is limited to the length of one line, i.e., 1024 characters.

10

Assembler Listing File

The assembly listing file is the output file of the Assembler that contains information about the generated code. The listing file is generated when the `-L` assembler option is activated. When an error is detected during assembling from the file, no listing file is generated.

The amount of information available depends upon the following assembler options:

- -L: Generate a listing file on page 153
- -Lc: No Macro call in listing file on page 160
- -Ld: No macro definition in listing file on page 163
- -Le: No Macro expansion in listing file on page 166
- -Li: Not included file in listing file on page 169

The information in the listing file also depends on following assembler directives:

- LIST - Enable Listing on page 314
- NOLIST - Disable Listing on page 325
- CLIST - List conditional assembly on page 286
- MLIST - List macro expansions on page 322

The format from the listing file is influenced by the following assembler directives:

- PLEN - Set Page Length on page 334
- LLEN - Set Line Length on page 316
- TABS - Set Tab Length on page 342
- SPC - Insert Blank Lines on page 341
- PAGE - Insert Page break on page 332
- NOPAGE - Disable Paging on page 327
- TITLE - Provide Listing Title on page 343.

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

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, e.g., HC12.
- The third line contains a copyright notice (Listing 10.1 on page 356).

Listing 10.1 Example page header output

```
Demo Application
Freescale HC12-Assembler
(c) COPYRIGHT Freescale 1991-2005
```

Source listing

The printed columns can be configured in various formats with the -Lasmc: Configure listing file on page 156 assembler option. The default format of the source listing has five columns:

- “Abs.” on page 356,
- “Rel.” on page 357,
- “Loc” on page 358,
- “Obj. code” on page 359, and
- “Source line” on page 360.

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 10.2 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                                 LDAA  \1
  12    3i                                 STAA  \2
  13    4i                               ENDM
  14   10                       CodeSec: SECTION
  15   11                       Start:
  16   12                                cpChar  char1, char2
  17    2m  000000 B6 xxxx    +            LDAA  char1
  18    3m  000003 7A xxxx    +            STAA  char2
  19   13   000006 A7                    NOP
  20   14   000007 A7                    NOP
```

In the previous example, the line number displayed in the 'Abs.' column is incremented for each line.

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 Listing 10.3 on page 357.

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 10.3 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                                LDAA \1
   12    3i                                STAA \2
   13    4i                              ENDM
```

```
  14    10                      CodeSec: SECTION
  15    11                      Start:
  16    12                               cpChar char1, char2
  17     2m   000000 B6 xxxx    +          LDAA char1
  18     3m   000003 7A xxxx    +          STAA char2
  19    13    000006 A7                  NOP
  20    14    000007 A7                  NOP
```

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 '`LDAA  char1`' is generated by a macro expansion.

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 Listing 10.4 on page 358.

Listing 10.4 Example Listing File - Loc column

```
 Abs. Rel.   Loc     Obj. code   Source line
 ---- ----   ---------- ---------   -----------------------------
    1    1                      ;-------------------------------
    2    2                      ; File: test.o
    3    3                      ;-------------------------------
    4    4
    5    5                               XDEF Start
    6    6                      MyData:  SECTION
    7    7   000000             char1:   DS.B  1
    8    8   000001             char2:   DS.B  1
    9    9                               INCLUDE "macro.inc"
   10    1i                     cpChar:  MACRO
   11    2i                                LDAA  \1
   12    3i                                STAA  \2
   13    4i                              ENDM
   14   10                      CodeSec: SECTION
```

```
  15    11                      Start:
  16    12                               cpChar char1, char2
  17     2m   000000 B6 xxxx     +         LDAA  char1
  18     3m   000003 7A xxxx     +         STAA  char2
  19    13    000006 A7                  NOP
  20    14    000007 A7                  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 ‘LDAA char1’ is located at offset 0 from the section ‘codeSec’ start address.

The instruction ‘STAA char2’ is located at offset 3 from the section ‘codeSec’ start address.

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 Listing 10.5 on page 359.

Listing 10.5 Example listing file - Obj. 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                                LDAA  \1
   12    3i                                STAA  \2
   13    4i                              ENDM
   14    10                     CodeSec: SECTION
   15   11                      Start:
   16   12                               cpChar char1, char2
```

```
  17    2m   000000 B6 xxxx    +          LDAA char1
  18    3m   000003 7A xxxx    +          STAA char2
  19   13    000006 A7                   NOP
  20   14    000007 A7                   NOP
```

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 Listing 10.6 on page 360.

Listing 10.6 Example listing file - Source line column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ;-------------------------------
    2    2                      ; File: test.o
    3    3                      ;-------------------------------
    4    4
    5    5                                XDEF Start
    6    6                      MyData:   SECTION
    7    7   000000             char1:    DS.B   1
    8    8   000001             char2:    DS.B   1
    9    9                                INCLUDE "macro.inc"
   10   1i                      cpChar:   MACRO
   11   2i                                  LDAA   \1
   12   3i                                  STAA   \2
   13   4i                                ENDM
   14   10                      CodeSec:  SECTION
   15   11                      Start:
   16   12                                cpChar char1, char2
   17    2m  000000 B6 xxxx    +           LDAA char1
   18    3m  000003 7A xxxx    +           STAA char2
   19   13   000006 A7                    NOP
   20   14   000007 A7                    NOP
```

11

Mixed C and Assembler Applications

When you intend to mix Assembly source file and ANSI-C source files in a single application, the following issues are important:

- “Memory models” on page 361
- “Parameter passing scheme” on page 362
- “Return Value” on page 363
- “Accessing assembly variables in an ANSI-C source file” on page 363
- “Accessing ANSI-C variables in an assembly source file” on page 364
- “Invoking an assembly function in an ANSI-C source file” on page 365
- “Support for structured types” on page 368

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.

Memory models

The memory models are only important if you mix C and assembly code. In this case all sources must be compiled or assembled with the same memory model.

The Assembler supports all memory models of the compiler. Depending on your hardware, use the smallest memory model suitable for your programming needs.

Table 11.1 on page 362 summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

Table 11.1 HC(S)12 Memory Models

Optio n	Memory Model	Local Data	Global Data	Suggested Use
–Ms	SMALL	SP rel	extended	Small applications which fit into the 64k address space or which do only have limited places where paged area is accessed.
–Mb	BANKED	SP rel	extended	Larger applications which code does not fit into the 64k address space. Data is limited to the 64k address space. The code generated by the compiler is not much larger than in the SMALL memory model because the CPU supports the CALL instruction. Usually there is one additional byte per function call.
–Ml	LARGE	SP rel	far	Applications whose data does not fit into 64k address space. The code generated by the compiler is significantly larger than in the other memory models.

NOTE The default pointer size for the compiler is also affected by the memory model chosen.

Parameter passing scheme

When you are using the HC12 compiler, the parameter passing scheme is the following:

The Pascal calling convention is used for functions with a fixed number of parameters: The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack again.

The C calling convention is used only for functions with a variable number of parameters. In this case the caller pushes the arguments from right to left.

If the last parameter of a function with a fixed number of arguments has a simple type, it is not pushed but passed in a register. This results in shorter code because pushing the last parameter can be avoided. Table 11.2 on page 363 shows an overview of the registers used for argument passing

Table 11.2 Registers used for passing the last argument to a function

Size of Last Parameter	Type example	Register
1 byte	char	B
2 bytes	int, array	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Parameters having a type not listed are passed on the stack (i.e., all those having a size greater than 4 bytes).

Return Value

Function results usually are returned in registers, except if the function returns a result larger than 4 bytes (see Table 11.3 on page 363). Depending on the size of the return type, different registers are used:

Table 11.3 Data type and registers used in function returns

Size of return value	Type example	Register
1 byte	char	B
2 bytes	int	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Functions returning a result larger than two words are called with an additional parameter. This parameter is the address where the result should get copied to.

Accessing assembly variables in an ANSI-C source file

A variable or constant defined in an assembly source file is accessible in an ANSI-C source file.

The variable or constant is defined in the assembly source file using the standard assembly syntax.

Variables and constants must be exported using the `XDEF` directive to make them visible from other modules (Listing 11.1 on page 364).

Listing 11.1 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 (Listing 11.2 on page 364).

Listing 11.2 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 (Listing 11.3 on page 364).

Listing 11.3 Example of data and constant reference

```
  ASMData = ASMConst + 3;
```

Accessing ANSI-C variables in an 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 (Listing 11.4 on page 365).

Listing 11.4 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 (Listing 11.5 on page 365).

This can also be done in a separate file, included in the assembly source file.

Listing 11.5 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 (Listing 11.6 on page 365).

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

Listing 11.6 Example of data and constant reference

```
  LDAA CConst
....
  LDAA CData
 ....
```

Invoking an assembly function in an ANSI-C source file

An function implemented in an assembly source file (`mixasm.asm` in Listing 11.7 on page 366) can be invoked in a C source file (Listing 11.9 on page 367). 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 11.7 Example of an assembly file: mixasm.asm

```
            XREF  CData
            XDEF  AddVar
            XDEF  ASMData

DataSec:    SECTION
ASMData:    DS.B  1
CodeSec:    SECTION
AddVar:
            ADDB  CData   ; add CData to the parameter in register B
            STAB  ASMData ; result of the addition in ASMData
            RTS
```

We recommend that you generate a header file for each assembly source file (Listing 11.7 on page 366). This header file (`mixasm.h` in Listing 11.8 on page 366) should contain the interface to the assembly module.

Listing 11.8 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, by using its name.

Example of a C file

A C source code file (`mixc.c`) has the main() function which calls the `AddVar()` function. See Listing 11.9 on page 367. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format).

Listing 11.9 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 (ASMData != CData + 10){
    Error = 1;
  } else {
    Error = 0;
  }
  for(;;); // wait forever
}
```

NOTE 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 Listing 11.10 on page 367.

Listing 11.10 Example of linker parameter file: mixasm.prm

```
LINK mixasm.abs
NAMES
  mixc.o mixasm.o
END
SECTIONS
  MY_ROM   = READ_ONLY  0x4000 TO 0x4FFF;
  MY_RAM   = READ_WRITE 0x2400 TO 0x2FFF;
  MY_STACK = READ_WRITE 0x2000 TO 0x23FF;
END
PLACEMENT
  DEFAULT_RAM     INTO MY_RAM;
  DEFAULT_ROM     INTO MY_ROM;
  SSTACK          INTO MY_STACK;
END

INIT main
```

NOTE We recommend that you use the same memory model and object file format for all the generated object files.

Support for structured types

When the -Struct: Support for structured types on page 186 assembler option is activated, the Macro Assembler also supports the definition and usage of structured types. This allows an easier way to access ANSI-C structured variable in the Macro Assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type. See “Structured type definition” on page 368.
- Define a structured variable. See “Variable definition” on page 370.
- Declare a structured variable. See “Variable declaration” on page 370.
- Access the address of a field inside of a structured variable. See “Accessing a field address” on page 371
- Access the offset of a field inside of a structured variable. See “Accessing a field offset” on page 372.

NOTE Some limitations apply in the usage of the structured types in the Macro Assembler. See Structured type: Limitations on page 373.

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 Listing 11.11 on page 368:

Listing 11.11 Definition for STRUCT

```
typeName: STRUCT
   lab1: DS.W 1
   lab2: DS.W 1
         ...
```

```
ENDSTRUCT
```

where:

- `'typeName'` is the name associated with the defined type. The type name is considered to be a user-defined keyword. The Macro Assembler will be case-insensitive on `typeName`.
- `'STRUCT'` specifies that the type is a structured type.
- `'lab1'`and `'lab2'` are the fields defined inside of the `'typeName' type`. The fields will be considered as user-defined labels, and the Macro Assembler will be case-sensitive on label names.

As with all other directives in the Assembler, the `STRUCT` and `UNION` directives are case-insensitive.

The `STRUCT` and `UNION` directives cannot start on column 1 and must be preceded by a label.

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.

Table 11.4 on page 369 shows how the ANSI-C standard types are converted in the assembler notation:

Table 11.4 Converting ANSI-C standard types to assembler notation

ANSI-C type	Assembler Notation
char	DS - Define Space on page 292
short	DS.W
int	DS.W
long	DS.L
enum	DS.W
bitfield	-- not supported --
float	-- not supported --*DS.F*

Table 11.4 Converting ANSI-C standard types to assembler notation (*continued*)

ANSI-C type	Assembler Notation
double	-- not supported --*DS.D*
data pointer	DS.W
function pointer	-- not supported --

Variable definition

The Macro Assembler can provide a way to define a variable with a specific type. This is done using the following syntax (Listing 11.12 on page 370):

`var: typeName`

where:

- `'var'` is the name of the variable.
- `'typeName'` is the type associated with the variable.

Listing 11.12 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’
```

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 Listing 11.13 on page 371 for an example.

Listing 11.13 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’
```

Accessing a structured variable

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

Accessing a field address

To access a structured-type field address (Listing 11.14 on page 371), 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 11.14 Example of accessing a field address

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
field3:    DS.B 1
field4:    DS.B 3
field5:    DS.W 1
         ENDSTRUCT

         XREF myData:myType
         XDEF entry
```

```
CodeSec: SECTION
entry:
         LDAA myData:field3 ; Loads register A with the
                            ; contents 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.

Accessing a field offset

To access a structured type field offset, the Assembler will use following notation:

`<typeName>-><field>`

where:

- '`typeName`' is the name of a structured type.
- '`field`' is the name of a field in the structured type associated with the variable. See Listing 11.15 on page 372 for an example of using this notation for accessing an offset..

Listing 11.15 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:
         LDX #myData
         LDAA myType->field3,X ; Adds the offset of field
                               ;'field3' (4) to X and loads
                               ; A with the content of the
                               ; effective address
```

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.

12

Make Applications

This chapters has the following sections:

- “Assembly applications” on page 375
- “Memory maps and segmentation” on page 376

Assembly applications

This section covers:

- Directly generating an absolute file on page 375
- Mixed C and assembly applications on page 375

Directly generating an absolute file

When an absolute file is directly generated by the Assembler:

- the application entry point must be specified in the assembly source file using the directive `ABSENTRY`.
- The whole application must be encoded in a single assembly unit.
- The application should only contain absolute sections.

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.

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.

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 (Listing 12.1 on page 376).

Listing 12.1 SECTIONS/PLACEMENT portion of a PRM file

```
SECTIONS
  ROM1        = READ_ONLY  0x0200 TO 0x0FFF;
  SpecialROM = READ_ONLY  0x8000 TO 0x8FFF;
  RAM         = READ_WRITE 0x4000 TO 0x4FFF;
END

PLACEMENT
  DEFAULT_ROM   INTO ROM1;
  mySection     INTO SpecialROM;
  DEFAULT_RAM   INTO RAM;
END
```

13

How to ...

This chapter covers the following topics:

- "How to work with absolute sections" on page 377
- "How to work with relocatable sections" on page 380
- "How to initialize the Vector table" on page 383
- "Splitting an application into different modules" on page 390
- "Using the direct addressing mode to access symbols" on page 392

How to work 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)

Defining absolute sections in an assembly source file

An absolute section is defined using the `ORG` directive. In that case, the Macro Assembler generates a pseudo section, whose name is "`ORG_<index>`", where index is an integer which is incremented each time an absolute section is encountered (Listing 13.1 on page 377).

Listing 13.1 Defining an absolute section containing data

```
        ORG    $800     ; Absolute data section.
var:    DS.    1
        ORG    $A00     ; Absolute constant data section.
cst1:   DC.B   $A6
cst2:   DC.B   $BC
```

In the previous portion of code, the `cst1` label is located at address `$A00`, and the `cst2` label is located at address `$A01`.

Listing 13.2 Assembler output listing for Listing 13.1 on page 377

```
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
```

Program assembly source code should be located in a separate absolute section (Listing 13.3 on page 378).

Listing 13.3 Defining an absolute section containing code

```
        XDEF   entry
        ORG    $C00   ; Absolute code section.
entry:
LDAA    cst1          ; Load value in cst1
        ADDA   cst2   ; Add value in cst2
        STAA   var    ; Store in var
        BRA    entry
```

In the previous portion of code, the instruction LDAA will be located at address $C00, and instruction ADDA at address $C03. See Listing 13.4 on page 378.

Listing 13.4 Assembler output listing for Listing 13.3 on page 378

```
   6    6                                        ORG    $C00 ; Absolute section.
   7    7                                 entry:
   8    8   a000C00 B6 0A00                      LDAA   cst1 ; Load value
   9    9   a000C03 BB 0A01                      ADDA   cst2 ; Add value in cst2
  10   10   a000C06 7A 0800                      STAA   var  ; Store in var
  11   11   a000C09 20F5                         BRA    entry
```

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 instruction `LDS` can be used to initialize the stack pointer.

- 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.

Linking an application containing absolute sections

When the Assembler is generating an object file, applications containing only absolute sections must be linked. The linker parameter file must contain at least:

- the name of the absolute file
- the name of the object file which should be linked
- the specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- the specification of a memory area where the sections containing code or constants must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- the specification of the application entry point, and
- the definition of the reset vector.

The minimal linker parameter file will look as shown in Listing 13.5 on page 379.

Listing 13.5 Minimal linker parameter file

```
LINK test.abs  /* Name of the executable file generated.      */
NAMES
  test.o       /* Name of the object files 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.
*/
  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 0xFFFE entry /* Initialization of the reset vector. */
```

CAUTION 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.

How to work with relocatable sections

A relocatable section is a section which start address is determined at linking time.

(See modules `fibo.asm` and `fibo.prm` in the demo directory)

Defining relocatable sections in a source file

A relocatable section is defined using the `SECTION` directive. See Listing 13.6 on page 380 for an example of defining relocatable sections.

Listing 13.6 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 Listing 13.7 on page 381.

Listing 13.7 Assembler output listing for Listing 13.6 on page 380

```
  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
```

Program assembly source code should be located in a separate relocatable section (Listing 13.8 on page 381).

Listing 13.8 Defining a relocatable section for code

```
         XDEF entry
codeSec: SECTION     ; Relocatable code section.
entry:
         LDAA  cst1  ; Load value in cst1
         ADDA  cst2  ; Add value in cst2
         STAA  var   ; Store in var
         BRA   entry
```

In the previous portion of code, the instruction LDAA will be located at offset 0 from the section codeSec start address, and instruction ADDA at offset 3 from the section `codeSec` 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 instruction LDS can be used to initialize the stack pointer.
- Publish the application's entry point using the `XDEF` directive.

Linking an application containing relocatable sections

Applications containing relocatable sections must be linked. The linker parameter file must contain at least:

- the name of the absolute file,
- the name of the object file which should be linked,
- the specification of a memory area where the sections containing variables must be allocated,

- the specification of a memory area where the sections containing code or constants must be allocated,
- the specification of the application's entry point, and
- the definition of the reset vector.

A minimal linker parameter file will look as shown in Listing 13.9 on page 382.

Listing 13.9 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 0xFFFE 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

The sample `main.asm` module created by a CodeWarrior's New Project Wizard relocatable assembly project is an example of usage of relocatable sections in an application.

How to initialize the 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.

- on page 383Initializing the Vector table in the linker PRM file on page 383 (recommended),
- Initializing the Vector table in a source file using a relocatable section on page 385, or
- Initializing the Vector table in a source file using an absolute section on page 388.

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 13.10 on page 383). All these labels must be published, otherwise they cannot be addressed in the linker `PRM` file.

Listing 13.10 Initializing the Vector table from a PRM File

```
         XDEF IRQFunc, XIRQFunc, SWIFunc, OpCodeFunc, ResetFunc
DataSec: SECTION
Data:    DS.W  5 ; Each interrupt increments an element in the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
         LDAB  #0
         BRA   int
XIRQFunc:
         LDAB  #2
         BRA   int
SWIFunc:
         LDAB  #4
         BRA   int
OpCodeFunc:
         LDAB  #6
         BRA   int
ResetFunc:
         LDAB  #8
         BRA   entry
int:
         LDX   #Data ; Load address of symbol Data in X
```

```
        ABX    ; X <- address of the appropriate element in the table
        INC    0, X   ; The table element is incremented
        RTI
entry:
        LDS #$AFE
loop:   BRA loop
```

NOTE The functions `'IRQFunc'`, `'XIRQFunc'`, `'SWIFunc'`, `'OpCodeFunc'`, `'ResetFunc'` are published. This is required because they are referenced in the linker PRM file.

NOTE As the processor automatically pushes all registers on the stack on occurrence of an interrupt, the interrupt function do not need to save and restore the registers it is using

NOTE All Interrupt functions must be terminated with an `RTI` instruction

The vector table is initialized using the linker `VECTOR ADDRESS` command (Listing 13.11 on page 384).

Listing 13.11 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;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END

INIT ResetFunc
VECTOR ADDRESS 0xFFF2 IRQFunc
VECTOR ADDRESS 0xFFF4 XIRQFunc
VECTOR ADDRESS 0xFFF6 SWIFunc
VECTOR ADDRESS 0xFFF8 OpCodeFunc
```

```
VECTOR ADDRESS 0xFFFE 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 0xFFF2 IRQFunc'` specifies that the address of the `'IRQFunc'` function should be written at address `0xFFF2`.

Initializing the Vector table in a source file using a relocatable section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions that should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See Listing 13.12 on page 385.

Listing 13.12 Initializing the Vector table in source code with a relocatable section

```
         XDEF   ResetFunc
DataSec: SECTION
Data:    DS.W   5   ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
         LDAB   #0
         BRA    int
XIRQFunc:
         LDAB   #2
         BRA    int
SWIFunc:
         LDAB   #4
         BRA    int
OpCodeFunc:
         LDAB   #6
         BRA    int
ResetFunc:
         LDAB   #8
         BRA    entry
```

```
DummyFunc:
         RTI
int:
         LDX   #Data
         ABX
         INC   0, X
         RTI
entry:
         LDS #$AFE
loop:    BRA loop

VectorTable:SECTION
; Definition of the vector table.
IRQInt:        DC.W  IRQFunc
XIRQInt:       DC.W  XIRQFunc
SWIInt:        DC.W  SWIFunc
OpCodeInt:     DC.W  OpCodeFunc
COPResetInt:   DC.W  DummyFunc  ; No function attached to COP Reset.
ClMonResInt:   DC.W  DummyFunc  ; No function attached to Clock
                                ; MonitorReset.
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'`.

NOTE In the previous example, the constant `'XIRQInt'` is initialized with the address of the label `'XIRQFunc'`.

NOTE All the labels specifying an initialization value must be defined, published (using `XDEF`), or imported (using `XREF`) in the assembly source file

The section should now be placed at the expected address. This is performed in the linker parameter file (Listing 13.13 on page 386).

Listing 13.13 Example linker parameter file

```
LINK  test.abs
NAMES test.o
END
```

```
SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
   /* Define the memory range for the vector table */
  Vector  = READ_ONLY  0xFFF2 TO 0xFFFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
   /* Place the section 'VectorTable' at the appropriated address. */
  VectorTable        INTO Vector;
END

INIT ResetFunc
ENTRIES
  *
END
```

NOTE The statement `'Vector = READ_ONLY 0xFFF2 TO 0xFFFF'` defines the memory range for the vector table.

NOTE The statement `'VectorTable INTO Vector'` specifies that the `VectorTable` section should be loaded in the read only memory area `Vector`. This means, the constant `'IRQInt'` will be allocated at address `0xFFF2`, the constant `'XIRQInt'` will be allocated at address `0xFFF4`, and so on. The constant `'ResetInt'` will be allocated at address `0xFFFE`.

NOTE The statement `'ENTRIES * END'` 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.

NOTE When developing a banked application, make sure that the code from the interrupt functions is located in the non banked memory area.

Initializing the Vector table in a source file using an absolute section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See Listing 13.14 on page 388 for an example.

Listing 13.14 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
; Implementation of the interrupt functions.
IRQFunc:
           LDAB  #0
           BRA   int
XIRQFunc:
           LDAB  #2
           BRA   int
SWIFunc:
           LDAB  #4
           BRA   int
OpCodeFunc:
           LDAB  #6
           BRA   int
ResetFunc:
           LDAB  #8
           BRA   entry

DummyFunc:
           RTI
int:
           LDX   #Data
           ABX
           INC   0, X
           RTI
entry:
           LDS  #$AFE
loop:      BRA   loop

           ORG   $FFF2
```

```
;Definition of the vector table
;in an absolute section starting at address $FFF2.
IRQInt:         DC.W IRQFunc
XIRQInt:        DC.W XIRQFunc
SWIInt:         DC.W SWIFunc
OpCodeInt:      DC.W OpCodeFunc
COPResetInt:    DC.W DummyFunc  ; No function attached to COP Reset.
ClMonResInt:    DC.W DummyFunc  ; No function attached to Clock
                                ; MonitorReset.
ResetInt:       DC.W ResetFunc
```

The section should now be placed at the expected address. This is performed in the linker parameter file (Listing 13.15 on page 389).

NOTE Each constant in the section starting at `$FFF2` is defined as a word (a 2-byte constant), because the entry in the vector table are 16 bits wide.

NOTE In the previous example, the constant `'IRQInt'` is initialized with the address of the label `'IRQFunc'`.

NOTE All the labels with an initialization value must be defined, published (using `XDEF`) or imported (using `XREF`) in the assembly source file.

NOTE The statement `'ORG $FFF2'` specifies that the following section must start at address `$FFF2`.

Listing 13.15 Example linker parameter file for Listing 13.14 on page 388:

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END
```

```
INIT ResetFunc
ENTRIES
  *
END
```

NOTE The statement 'ENTRY * END' 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.

NOTE When developing a banked application, make sure that the code from the interrupt functions is located in the non-banked memory area

Splitting an application into different 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 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.

Example of an assembly file (Test1.asm)

See Listing 13.16 on page 390 for an example assembly file which is used in the following sections.

Listing 13.16 Separating Code into Modules—Test1.asm

```
           XDEF AddSource
           XDEF Source

initStack: EQU $AFF

DataSec:   SECTION
Source:    DS.B 1
CodeSec:   SECTION
AddSource:
```

```
          ADDA Source
          STAA Source
          RTS
```

Corresponding include file (Test1.inc)

See Listing 13.17 on page 391 for an example Test1inc include file.

Listing 13.17 Separating Code into Modules—Test1.inc

```
         XREF AddSource
; The AddSource function adds the value stored in the variable
; Source to the contents of the A register. The result of the
; computation is stored in the Source variable.
;
; Input Parameter:  The A register contains the value that should be
;                   added to the Source variable.
; Output Parameter: Source contains the result of the addition.

         XREF Source
; The Source variable is a 1-byte variable.
```

Example of an assembly File (Test2.asm)

Listing 13.18 on page 391 is another assembly code file module for this project.

Listing 13.18 Separating Code into Modules—Test2.asm

```
           XDEF entry
           INCLUDE "Test1.inc"

initStack: EQU $AFE

CodeSec:   SECTION
entry:     LDS #initStack
           LDAA #$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.

Example of a PRM file (Test2.prm)

Listing 13.19 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

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 0xFFFE 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.

Using the direct addressing mode to access symbols

There are different ways for the Assembler to use the direct addressing mode on a symbol:

- "Using the direct addressing mode to access external symbols" on page 393,
- "Using the direct addressing mode to access exported symbols" on page 393,

- “Defining symbols in the direct page” on page 394,
- “Using the force operator” on page 394, or
- “Using SHORT sections” on page 395.

Using the direct addressing mode to access external symbols

External symbols, which should be accessed using the direct addressing mode, must be declared using the `XREF.B` directive. Symbols which are imported using `XREF` are accessed using the extended addressing mode.

Listing 13.20 Using direct addressing to access external symbols

```
     XREF.B ExternalDirLabel
     XREF   ExternalExtLabel

     LDD    ExternalDirLabel ; Direct addressing mode is used.

     LDD    ExternalExtLabel ; Extended addressing mode is used.
```

Using the direct addressing mode to access exported symbols

Symbols, which are exported using the `XDEF.B` directive, will be accessed using the direct addressing mode. Symbols which are exported using `XDEF` are accessed using the extended addressing mode.

Listing 13.21 Using direct addressing to access exported symbols

```
XDEF.B DirLabel
XDEF   ExtLabel
…
LDD   DirLabel ; Direct addressing mode is used.
…
LDD   ExtLabel ; Extended addressing mode is used.
```

Defining symbols in the direct page

Symbols that are defined in the predefined `BSCT` section are always accessed using the direct-addressing mode (Listing 13.22 on page 394).

Listing 13.22 Defining symbols in the direct page

```
…
           BSCT
DirLabel: DS.B   3
dataSec:  SECTION
ExtLabel: DS.B   5
…
codeSec:  SECTION
…
          LDD   DirLabel ; Direct addressing mode is used.
…
          LDD   ExtLabel ; Extended addressing mode is used.
```

Using the force operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode (Listing 13.23 on page 394).

The supported force operators are:

- < or `.B` to force direct addressing mode
- > or `.W` to force extended addressing mode.

Listing 13.23 Using a force operator

```
…
dataSec: SECTION
label:   DS.B   5
…
codeSec: SECTION
…
         LDD   <label   ; Direct addressing mode is used.
         LDD   label.B ; Direct addressing mode is used.
…
         LDD   >label   ; Extended addressing mode is used.
         LDD   label.W ; Extended addressing mode is used.
```

Using SHORT sections

Symbols that are defined in a section defined with the `SHORT` qualifier are always accessed using the direct addressing mode (Listing 13.24 on page 395).

Listing 13.24 Using SHORT sections

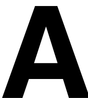
```
…
shortSec: SECTION SHORT
DirLabel: DS.B   3
dataSec:  SECTION
ExtLabel: DS.B   5
…
codeSec:  SECTION
…
          LDD   DirLabel ; Direct addressing mode is used.
…
          LDD   ExtLabel ; Extended addressing mode is used.
```

II

Appendices

This document has the following appendices:

- “Global Configuration File Entries” on page 399
- “Local Configuration File Entries” on page 409
- “MASM Compatibility” on page 429
- “MCUasm Compatibility” on page 433
- “Semi-Avocet Compatibility” on page 435



Global Configuration File Entries

This appendix documents the sections and entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [Installation] Section on page 400
- [Options] Section on page 401
- [XXX_Assembler] Section on page 402
- [Editor] Section on page 405

[Installation] Section

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
```

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
```

[Options] Section

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 on page 107 environment variable.

Example

```
DefaultDir=C:\install\project
```

[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 HC12 Assembler, the name of this section would be [HC12_Assembler].

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.

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 check box '*Appearance*' in the' Save Configuration' dialog box.

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.

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 check box '*Options*' in the 'Save Configuration' dialog box.

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
```

[Editor] Section

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.

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.

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

Example

Listing A.1 on page 407 shows a typical `mcutools.ini` file.

Listing A.1 Typical mcutools.ini file layout

```
[Installation]
Path=c:\Freescale
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\WinEdit32\WinEdit.exe
editor_opts=%f /#:%l

[Options]
DefaultDir=c:\myprj

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

B

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 could contain these sections for using the Assembler:

- [Editor] Section on page 409
- [XXX_Assembler] Section on page 413

See the Example on page 428 section for a sample project.ini file.

[Editor] Section

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.

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.

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=WINEdit
editor_exe=C:\WinEdit32\WinEdit.exe
editor_opts=%f /#:%l
```

[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 HC12 Assembler, the name of this section would be `[HC12_Assembler]`.

RecentCommandLineX, X= integer

Arguments

String with a command line history entry, e.g., `fibo.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.

CurrentCommandLine

Arguments

String with the command line, e.g., "`fibo.asm -w1`"

Description

The currently visible command line content.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

StatusbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it 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.

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.

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,..) and other flags.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

WindowFont

Arguments

`size:` == 0 -> generic size, < 0 -> font character height, > 0 -> font cell height

`weight`: 400 = normal, 700 = bold (valid values are 0..1000)

`italic`: 0 == no, 1 == yes

`font name`: max. 32 characters.

Description

Font attributes.

Saved

Only with *Appearance* set in the *File* > *Configuration Save Configuration* dialog box.

Example

```
WindowFont=-16,500,0,Courier
```

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.

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.

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.

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.

EditorCommandLine

Arguments

command line, for `WinEdit: "C:\WinEdit32\WinEdit.exe %f /#:%l"`

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.

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.

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.

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.

Example

The example in Listing B.1 on page 428 shows a typical layout of the configuration file (usually `project.ini`).

Listing B.1 Example of a project.ini file

```
[Editor]
Editor_Name=IDF
Editor_Exe=C:\WinEdit32\WinEdit.exe
Editor_Opts=%f /#:%l

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

C

MASM Compatibility

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

Comment Line

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

Constants (Integers)

For compatibility with the MASM Assembler, the following notations are also supported for integer constants (Listing C.1 on page 429):

- 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 C.1 Integer examples

```
512d        ; decimal representation
512D        ; decimal representation
200h        ; hexadecimal representation
200H        ; hexadecimal representation
1000o       ; octal representation
1000O       ; octal representation
1000q       ; octal representation
1000Q       ; octal representation
1000000000b ; binary representation
1000000000B ; binary representation
```

Operators

For compatibility with the MASM Assembler, the following notations in Table C.1 on page 430 are also supported for operators:

Table C.1 Operator notation for MASM compatibility

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

Directives

Table C.2 on page 430 enumerates the directives that are supported by the Macro Assembler for compatibility with MASM:

Table C.2 Supported MASM directives

Operator	Notation	Description
RMB	DS	Define storage for a variable. Argument specifies the byte size
RMD	DS 2*	Define storage for a variable. Argument specifies the number of 2-byte blocks
RMQ	DS 4*	Define storage for a variable. Argument specifies the number of 4-byte blocks
ELSEC	ELSE	Alternate of conditional block
ENDC	ENDIF	End of conditional block
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.

Table C.2 Supported MASM directives (*continued*)

Operator	Notation	Description
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 the switching to a section which has been defined previously.
ASCT		Creates a predefined section which name id ASCT.
BSCT		Creates a predefined section which name id BSCT. Variable defined in this section are accessed using the direct addressing mode.
CSCT		Creates a predefined section which name id CSCT.
DSCT		Creates a predefined section which name id DSCT.
IDSCT		Creates a predefined section which name id IDSCT.
IPSCT		Creates a predefined section which name id IPSCT.
PSCT		Creates a predefined section which name id PSCT.

D

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 covres the following topics:

- “Labels” on page 433
- “SET directive” on page 433
- “Obsolete directives” on page 434

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 D.1 on page 433).

Listing D.1 Erroneous label for MCUasm compatibility

```
label       DC.B   1
```

When MCUasm compatibility mode is not activated, the previous portion of code does not generate any error message.

SET directive

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 D.2 on page 434):

Listing D.2 SET directive

```
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 absolute expressions.

Obsolete directives

Table D.1 on page 434 enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.:

Table D.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.

E

Semi-Avocet Compatibility

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

Avocet compatibility mode can be activated, specifying the -C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON on page 132 assembler option. The compatibility does not cover all specific Avocet features but only some of them.

- "Directives" on page 435
- "Section Definition" on page 437
- "Macro parameters" on page 439
- "Support for Structured Assembly" on page 439.

Directives

Table E.1 on page 435 enumerates the directives which are supported when the Avocet Assembler compatibility mode is activated.

Table E.1 Avocet Assembler directives

Directive	Notation	Description
DEFSEG		Segment definition (See the "Section Definition" section below).

ELSEIF		Conditional directive, checking a specific condition.
		`IF ((label1 & label2) != 0)`
		`  LDD   #label1`
		`ELSIF (label1 = 0)`
		`  LDD   #label2`
		`ELSE`
		`  LDD   #0`
		`ENDIF`

Table E.1 Avocet Assembler directives (*continued*)

Directive	Notation	Description

EXITM	MEXIT	Define an exit condition for a macro.
		`Copy      MACRO source, dest`
		`            IFB "source"`
		`              EXITM`
		`            ENDIF`
		`            LDD   source`
		`            STD   dest`
		`          ENDM`

IFB Param	IFC Param, ““	Test if a macro parameter is empty. The syntax is IFB "param".
		`Copy      MACRO   source, des`
		`            IFB   "source"`
		`              LDD   #0`
		`              STD   dest`
		`            ELSE`
		`              LDD   source`
		`              STD   dest`
		`            ENDIF`
		`          ENDM`

IFNB Param	IFNC Param ““	Test if a macro parameter is not empty. The syntax is IFNB "param"
		`Copy        MACRO   source, dest`
		`              IFNB  "source"`
		`              LDD   source`
		`              STD   dest`
		`            ELSE`
		`              LDD   #0`
		`              STD   dest`
		`            ENDIF`
		`          ENDM`

NOSM	MLIST OFF	Do not insert the macro expansion in the listing file.
SEG	SWITCH	Switch to a previously defined segment. See the “Section Definition on page 437” section below.
SM	MLIST ON	Insert the macro expansion in the listing file.

Table E.1 Avocet Assembler directives (*continued*)

Directive	Notation	Description
SUBTITLE		Defines a subtitle for the input file. This subtitle is written to the listing file. `SUBTITLE title2: Main File`
TEQ	SET	Define a constant, which value may be modified in the source file.

Section Definition

Section definition is performed using the `DEFSEG` directive. The correct syntax for a `DEFSEG` directive is:

```
DEFSEG <name> [START=<start address>] [<section qualifier>]
```

where:

- `name`: is the name of the section
- `start address`: is the start address for the section. This parameter is optional.
- `section qualifier`: is the qualifier which applies to the section. This parameter is optional and may take the value:

Table E.2 section qualifiers

Qualifier	Meaning
PAGE0	for a data section located on the direct page
DATA	for a data section
CODE	for a code section

Some examples of the DEFSEG directive are shown in Listing E.1 on page 437.

Listing E.1 DEFSEG examples

```
DEFSEG myDataSection
DEFSEG D_ATC_TABLES START=$0EA0
DEFSEG myDirectData PAGE0
```

NOTE Because of an incompatibility in the object file format, an absolute section implementation must reside entirely in a single assembly unit. You cannot split the code from an absolute section over several object files. An absolute section is a section associated with a `start address`.

NOTE In order to split a section over different assembly units, you should define the section as relocatable (without `START`) and specify the address where you want to load the section in the linker PRM file.

The assembly source code in Listing E.2 on page 438 relates to a possible allocation of memory as shown in Listing E.3 on page 438.

Listing E.2 Example assembly code

```
DEFSEG D_ATC_TABLES ; START=$0EA0
```

Listing E.3 Portion of a linker parameter file

```
...
SECTION
   ...
  MY_TABLE = READ_WRITE 0x0EA0 TO 0x0EFF;
PLACEMENT
...
D_ATC_TABLES INTO MY_TABLE:
...
```

The `SEG` directive is then used to activate the corresponding section in the assembly source file.

The name specified in a `SEG` directive was previously specified in a `DEFSEG` directive.

The following syntax is acceptable for using the `SEG directive`:

```
SEG <name>
```

where:

`name:` is the name of the section, which was previously defined in a `DEFSEG` directive (Listing E.4 on page 439).

Listing E.4 Example of using the SEG directive

```
  SEG myDataSection
```

Macro parameters

When Avocet Compatibility is switched ON, names can be associated with macro parameters. A macro definition could be as in Listing E.5 on page 439:

Listing E.5 Example macro definition

```
Copy       MACRO   source, destination
             LDD   source
             STD   destination
           ENDM
```

Support for Structured Assembly

When the Avocet compatibility is switched ON, SWITCH, or FOR construct are available in Macro Assembler.

SWITCH block

The SWITCH directive evaluates an expression and assembles the code following the particular CASE statement which satisfies the switch expression. If no CASE statement corresponds to the value of the expression, the code following the DEFAULT (if present) is assembled.

ENDSW terminates the SWITCH directive.

The expression specified in a SWITCH directive must be an absolute expression (Listing E.6 on page 439).

Listing E.6 Example of using a SWITCH block

```
xxx    equ    5
  …
  SWITCH xxx
    CASE 0
      LDD    #1
```

```
    CASE 1
      LDD    2
    CASE 3
      LDD    #6
    DEFAULT
      LDD #0
  ENDSW
```

The instructions in Listing E.7 on page 440 are generated by the code in Listing E.6 on page 439. (Assuming that the value for `xxx` was still 5 when the `SWITCH` statement was encountered) there was no particular result for `xxx` equal to 5, so the result for the `DEFAULT CASE` ensues -
`LDD    #0.`

Listing E.7 Result of the SWITCH statement when xxx = 5

```
xxx    equ    5
  …
  LDD    #0
```

FOR Block

In the Avocet compatibility mode, the FOR - Repeat assembly block on page 307 assembler directive is supported (Listing E.8 on page 440).

Listing E.8 Example

```
  FOR l=2 TO 6
    NOP
  ENDFOR
```

The following instructions (Listing E.9 on page 440) are generated by the code segment in Listing E.8 on page 440.

Listing E.9

```
     NOP
     NOP
     NOP
     NOP
```

```
NOP
```

Index

Symbols
$() 99
${} 99
%(ENV) 126
%" 126
%' 126
%E 125
%e 126
%f 126
%N 125
%n 125
%p 125
* 272
.abs 120
.asm 119
.dbg 121
.hidefaults 97, 98
.inc 119
.ini 79
.lst 121
.o 120
.s1 120
.s2 120
.s3 120
.sx 120
{Compiler} 99
{Project} 99
{System} 99

A
About Box 92
ABSENTRY 63, 278
Absolute Expression 272, 273
Absolute Section 226, 231
ABSPATH 88, 104, 120
Addressing Mod 247
Addressing Mode
  Direct 249
  Extended 250
  Global 258
  Immediate 248
  Indexed 16-bit Offset 253
  Indexed 5-bit Offset 251
  Indexed 9-bit Offset 252
  Indexed Accumulator Offset 257
  Indexed Indirect 16-bit Offset 253
  Indexed Indirect D Accumulator Offset 257
  Indexed PC, Indexed PC Relative 258
  Indexed post-decrement 255
  Indexed post-increment 256
  Indexed pre-decrement 254
  Indexed pre-increment 255
  Inherent 248
  Relative 250
Addressing Modes 246
ALIGN 284, 301, 318
ASMOPTIONS 105
Assembler
  Configuration 79
  Error Feedback 93
  Input File 92
  Menu 80
  Menu Bar 78
  Messages 89
  Option 89
  Options Setting Dialog 89
  Output Files 119
  Status Bar 78
  Tool Bar 77
Assembler Option Settings dialog box 41, 68
assembler-output listing file 36
Avocet
  Directive
    DEFSEG 435, 436
    ELSEIF 435
    EXITM 436
    SEG 436
    SUBTITLE 437
    TEQ 437
  Macro Parameters 439
  Section Definition 437
  Structured Assembly 439

B

BASE 279, 285
Build Tool Utilities 18

C

-C=SAvocet 132
-Ci 133
CLIST 279
-CMacAngBrack 135
-CMacBrackets 136
CODE 125, 177
Code Section 225
CodeWarrior 86
CodeWarrior Development Studio 18
CodeWarrior project window 27
color 196, 197, 198, 199, 200
COM 86
Comment 259
-Compat 137
Complex Relocatable Expression 272
Constant
  Binary 262, 429
  Decimal 262, 429
  Floating point 263
  Hexadecimal 262, 429
  Integer 262
  Octal 262, 429
  String 263
Constant Section 225
Context menu 45
COPYRIGHT 106
-CpuHC12 143
-CpuHCS12 143
-CpuHCS12X 143
CTRL-S 88
Current Directory 98, 107
CurrentCommandLine 415

D

-D 146
Data Section 226
DC 288
DCB 290
Debug File 121, 314
Default Directory 401
DEFAULTDIR 107, 119
DefaultDir 401
DEFSEG 435, 436
Directive 246
  ABSENTRY 278
  ALIGN 284, 301, 318
  BASE 279, 285
  CLIST 279
  DC 288
  DCB 290
  DS 278, 292
  ELSE 294
  ELSEC 430
  END 296
  ENDC 430
  ENDFOR 279, 297
  ENDIF 281, 297, 298, 308
  ENDM 280, 320
  EQU 277, 300
  EVEN 301
  EXTERNAL 431, 434
  FAIL 279, 303
  FOR 307
  GLOBAL 431, 434
  IF 309, 311
  IFC 281, 311
  IFDEF 281, 312
  IFEQ 281, 311
  IFGE 281, 311
  IFGT 281, 311
  IFLE 281, 311
  IFLT 281, 311
  IFNC 281, 311
  IFNDEF 281, 312
  IFNE 281, 311
  INCLUDE 313
  LIST 279, 314
  LLEN 316
  LONGEVEN 279, 318
  Macro 319
  MEXIT 320
  MLIST 322

NOL 430, 434
NOLIST 280, 325
NOPAGE 280, 327
OFFSET 328
ORG 330
PAGE 280, 332
PLEN 334
PUBLIC 431, 434
RAD50 278, 335
RMB 430, 434
Section 277, 338
SET 340
SPC 341
TABS 342
TITLE 343
TTL 430, 434
XDEF 344
XREF 261, 278, 345
XREFB 278, 346, 431
DS 278, 292

E

Editor 409
Editor_Exe 405, 411
Editor_Name 405, 410
Editor_Opts 406, 412
EditorCommandLine 424
EditorDDEClientName 425
EditorDDEServiceName 427
EditorDDETopicName 426
EditorType 423
EDOUT 121
ELSE 294
ELSEC 430
ELSEIF 435
END 296
ENDC 430
ENDFOR 279, 297
ENDIF 281, 297, 298, 308
ENDM 280, 320
-ENV 148
ENVIRONMENT 108
Environment
ABSPATH 104, 120
ASMOPTIONS 105
COPYRIGHT 106
DEFAULTDIR 107, 119
ENVIRONMENT 108
ENVIRONMENT 97, 98
ERRORFILE 109
File 97
GENPATH 112, 119, 313
HIENVIRONMENT 108
INCLUDETIME 113
OBJPATH 114, 120
TEXTPATH 116
TMP 117
Variable 97
Environment Variable 103
ABSPATH 120
SRECORD 120
Environment Variables 88, 97
EQU 277, 300
Error File 121
Error Listing 121
ERRORFILE 109
EVEN 301
EXITM 436
Explorer 98
Expression 272
Absolute 272, 273
Complex Relocatable 272
Simple Relocatable 272, 274
EXTERNAL 431, 434
External Symbol 261

F

-F2 149
-F2o 149
-FA2 149
-FA2o 149
FAIL 279, 303
-Fh 149
Fibonacci series 19
File
Debug 121, 314
Environment 97
Error 121

Include 119
Listing 120, 121, 279, 314
Object 120
PRM 227, 229, 230
Source 119
File Manager 98
Floating-Point Constant 263
FOR 307

G

GENPATH 48, 71, 88, 112, 119, 313
GLOBAL 431, 434
Group 400
groups, CodeWarrior 27, 31, 32
GUI Graphic User Interface 73

H

-H 151
HC(S)12 Simulator 59
HIENVIRONMENT 108
HIGH 262
HOST 125

I

-I 152
IDF 98
IF 309, 311
IFC 281, 311
IFDEF 281, 312
IFEQ 281, 311
IFGE 281, 311
IFGT 281, 311
IFLE 281, 311
IFLT 281, 311
IFNC 281, 311
IFNDEF 281, 312
IFNE 281, 311
INCLUDE 313
Include Files 119
INCLUDETIME 113
Instruction 234
Integer Constant 262

L

-L 153
Label 234
LANGUAGE 125
-Lasmc 156
-Lasms 159
-Lc 161
-Ld 164
-Le 167
-Li 170
LIBPATH 88
-Lic 172, 174
-LicA 173
-LicWait 176
Line Continuation 102
Linker map file 36
Linker PRM file 50, 54
LIST 279, 314
Listing File 120, 121, 279, 314
LLEN 316
LONGEVEN 279, 318
LOW 262

M

Macro 246, 319
-MacroNest 178
-Mb 177
-MCUasm 179
MCUTOOLS.INI 99, 107
MESSAGE 125
Message Settings 89
MEXIT 320
-Ml 177, 362
MLIST 322
-Ms 362
-Mx 362

N

-N 180
New dialog box 20
New Project Wizard 19
-NoBeep 181
-NoDebugInfo 182

-NoEnv 183
NOL 430, 434
NOLIST 280, 325
NOPAGE 280, 327

O

Object File 120
-ObjN 184
OBJPATH 88, 114, 120
OFFSET 328
Operand 246
Operator 263, 430
- Addition 263, 271, 275
- Arithmetic AND 430
- Arithmetic Bit 275
- Arithmetic OR 430
- Arithmetic XOR 430
- Bitwise 266
- Bitwise (unary) 267
- Bitwise AND 272
- Bitwise Exclusive OR 272
- Bitwise OR 272
- Division 264, 271, 275
- Force 270
- HIGH 262, 269
- Logical 267
- LOW 262, 269
- Modulo 264, 271, 275
- Multiplication 264, 271, 275
- PAGE 262, 270
- Precedence 271
- Relational 268, 272
- Shift 265, 272, 275
- Shift left 430
- Shift right 430
- Sign 265, 271, 275
- Subtraction 264, 271, 275

Option
- CODE 125, 177
- HOST 125
- LANGUAGE 125
- MESSAGE 125
- OUTPUT 125
- VARIOUS 125

Options 401, 422
ORG 63, 330
OUTPUT 125

P

PAGE 262, 280, 332
PATH 114
Path 400
Path List 101
PLEN 334
PRM File 227, 229, 230
-Prod 186
project.ini 101
PUBLIC 431, 434

R

RAD50 278, 335
RecentCommandLine 414
Relocatable Section 228
Reserved Symbol 262
RGB 196, 197, 198, 199, 200
RMB 430, 434

S

SaveAppearance 402
SaveEditor 403
SaveOnExit 402
SaveOptions 403
Section 277, 338
- Absolute 226, 231
- Code 225
- Constant 225
- Data 226
- Relocatable 228

Sections 225
SEG 436
Select File to Assemble dialog box 44, 69
Select File to Link dialog box 57
SET 340
SHORT 339
ShowTipOfDay 421
Simple Relocatable Expression 272, 274
Source File 119

SPC 341
Special Modifiers 125
S-Record File 36
Starting 73
startup 101
StatusbarEnabled 416
String Constant 263
-Struct 187
SUBTITLE 437
Symbol 260
  External 261
  Reserved 262
  Undefined 261
  User Defined 260

T

TABS 342
TEQ 437
TEXTPATH 88, 116
Tip of the Day 73
TipFilePos 420
TITLE 343
TMP 117
ToolbarEnabled 417
TTL 430, 434

U

Undefined Symbol 261
UNIX 98
User Defined Symbol 260

V

-V 188
Variable
  Environment 97
VARIOUS 125
-View 189

W

-W1 191
-W2 192
-WErrFile 193
WindowFont 419
WindowPos 418
Windows 98
WinEdit 110
-Wmsg8x3 194
-WmsgCE 196
-WmsgCF 197
-WmsgCI 198
-WmsgCU 199
-WmsgCW 200
-WmsgFb 94
-WmsgFbiv 204
-WmsgFbm 201
-WmsgFbv 201
-WmsgFi 94, 195
-WmsgFim 204
-WmsgFob 206
-WmsgFoi 208
-WmsgFonp 203, 205, 207, 209, 211, 212, 213
-WmsgNe 214, 218
-WmsgNi 214, 215
-WmsgNu 216
-WmsgNw 214, 215, 218
-WmsgSd 219
-WmsgSe 220
-WmsgSi 221
-WmsgSw 222
-WOutFile 223
-WStdout 224

X

XDEF 344
XREF 261, 278, 345
XREFB 278, 346, 431