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

Revised: 05 January 2012

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, Texas 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

I Using HC(S)08/RS08 Assembler

Highlights 15
Structure of this Document 15

4 Working with Assembler 17

Programming Overview 17
Project Directory 18
External Editor 19
Managing Assembly Language Project Using CodeWarrior IDE 19
Create New Project. 20
Additional Project Information 21
Analysis of Groups and Files in a Project. 26
CodeWarrior Groups 26
Creating New Group 27
Adding New File to the Project. 27
Renaming File or Group 27
Moving File. 28
Removing File. 28
Restoring Deleted File. 28
Using Editor 28
Generating Listing Files 29
Writing your Assembly Source Files 31
Analyzing Project Files. 32
Assembling Source Files 36
Assembling and Linking with CodeWarrior IDE 36
Assembling with Assembler 36
Linking Application 51
Linking with CodeWarrior IDE 51
Linking with Linker 55
Directly Generating ABS File. 60

Creating Absolute Assembly Project 60
Adapting Absolute Assembly File Created by Wizard 61
Generating Absolute Assembly Using CodeWarrior IDE. 64
Generating Absolute Assembly Using Assembler Build Tool 65
Assembler Build Properties Panels. 66
HCS08 Assembler Build Properties Panels 68
HCS08 Assembler. 69
RS08 Assembler Build Properties Panels 89
RS08 Assembler 90

5 Assembler Graphical User Interface 111

Starting Assembler 112
Assembler Main Window 113
Window Title. 113
Content Area 114
Toolbar 115
Status Bar 115
Assembler Menu Bar. 115
File Menu 116
Assembler Menu 117
View Menu 117
Editor Setting Dialog Box. 118
Global Editor (shared by all tools and projects) 118
Local Editor (shared by all tools) 119
Editor Started with Command Line 120
Editor Started with DDE 121
CodeWarrior with COM 122
Modifiers. 123
Save Configuration Dialog Box 124
Environment Configuration Dialog Box. 126
Option Settings dialog box 127
Message Settings Dialog Box. 128
Changing the Class Associated with a Message 130
About Dialog Box. 131
Specifying Input File 132

Use Command Line in Toolbar to Assemble 132
Assembling a New File 132
Assembling a File which has Already been Assembled 132
Use File > Assemble Entry 132
Use Drag and Drop 132
Message/Error Feedback. 133
Use Information from Assembler Window 133
Use User-defined Editor 134
Line Number can be Specified on the Command Line 134
Line Number cannot be Specified on the Command Line 134

6 Environment 135

Current directory. 136
Environment macros 137
Global initialization file - mctools.ini (PC only) 138
Local configuration file (usually project.ini) 138
Line continuation 140
Environment variables details. 141
ABSPATH: Absolute file path. 142
ASMOPTIONS: Default assembler options 142
COPYRIGHT: Copyright entry in object file 143
DEFAULTDIR: Default current directory. 144
ENVIRONMENT: Environment file specification. 145
ERRORFILE: Filename specification error 146
GENPATH: Search path for input file. 148
INCLUDETIME: Creation time in the object file 149
OBJPATH: Object file path. 150
SRECORD: S-Record type 150
TEXTPATH: Text file path 151
TMP: Temporary directory 152
USERNAME: User Name in object file 153

7 Files 155

Input files 155
Source files 155

Include files 155
Output files 156
Object files 156
Absolute files. 156
S-Record Files 156
Listing files 157
Debug listing files 157
Error listing file 157
File processing 158

8 Assembler Options 159

Types of assembler options 159
Assembler Option details 161
Using special modifiers 162
List of Assembler option. 165
Detailed listing of all assembler options 168
-ArgFile: Specify a file from which additional command line options will be read 168
-AsmDbg: Emit assembly source file information in debug sections 168
-Ci: Switch case sensitivity on label names OFF 169
-CMacAngBrack: Angle brackets for grouping Macro Arguments 170
-CMacBrackets: Square brackets for macro arguments grouping. 171
-Compat: Compatibility modes 172
-CS08/-C08/-CRS08: Derivative family 174
-D: Define Label 175
-DefLabel: Improves support for data allocation directives 177
-Env: Set environment variable. 179
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format 179
-H: Short Help 181
-I: Include file path 182
-L: Generate a listing file 182
-Lasmc: Configure listing file 185
-Lasms: Configure the address size in the listing file 187
-Lc: No Macro call in listing file. 189
-Ld: No macro definition in listing file 191

-Le: No Macro expansion in listing file 193
-Li: No included file in listing file 195
-Lic: License information 197
-LicA: License information about every feature in directory 198
-LicBorrow: Borrow license feature 199
-LicWait: Wait until floating license is available from floating License Server 200
-Ll: Show label statistics 201
-M (-Ms, -Mt): Memory model 202
-MacroNest: Configure maximum macro nesting. 203
-MCUasm: Switch compatibility with MCUasm ON. 204
-MMU: Enable Memory Management Unit (MMU) Support 205
-N: Display notify box. 205
-NoBeep: No beep in case of an error 206
-NoDebugInfo: No debug information for ELF/DWARF files 207
-NoEnv: Do not use environment 208
-ObjN: Object filename specification 208
-Prod: Specify project file at startup 209
-Struct: Support for structured types 210
-V: Prints the Assembler version. 211
-View: Application standard occurrence 212
-W1: No information messages. 213
-W2: No information and warning messages 214
-WErrFile: Create "err.log" error file 215
-Wmsg8x3: Cut filenames in Microsoft format to 8.3 216
-WmsgCE: RGB color for error messages 217
-WmsgCF: RGB color for fatal messages. 217
-WmsgCI: RGB color for information messages 218
-WmsgCU: RGB color for user messages. 219
-WmsgCW: RGB color for warning messages 219
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode 221
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode 223
-WmsgFob: Message format for batch mode 224

-WmsgFoi: Message format for interactive mode. 226
-WmsgFonf: Message format for no file information. 228
-WmsgFonp: Message format for no position information. 229
-WmsgNe: Number of error messages 230
-WmsgNi: Number of Information messages 231
-WmsgNu: Disable user messages 232
-WmsgNw: Number of Warning messages. 233
-WmsgSd: Setting a message to disable 234
-WmsgSe: Setting a message to Error. 235
-WmsgSi: Setting a message to Information. 236
-WmsgSw: Setting a Message to Warning 237
-WOutFile: Create error listing file. 238
-WStdout: Write to standard output 239

9 Sections 241

Section attributes. 241
Code sections. 241
Constant sections. 241
Data sections 242
Section types 242
Absolute sections. 242
Relocatable sections 244
Relocatable vs. absolute sections 247
Modularity. 247
Multiple developers. 247
Early development. 248
Enhanced portability 248
Tracking overlaps 248
Reusability. 248

10 Assembler Syntax 249

Comment line 249
Source line 249
Label field 250
Operation field. 250

Operand field: Addressing modes (HC(S)08) 266
Operand Field: Addressing Modes (RS08) 277
Comment Field 280
Symbols 281
User-defined symbols 281
External symbols 282
Undefined symbols 282
Reserved symbols 283
Constants 283
Integer constants 283
String constants 284
Floating-Point constants 284
Operators 284
Addition and subtraction operators (binary) 285
Multiplication, division and modulo operators (binary) 285
Sign operators (unary) 286
Shift operators (binary) 287
Bitwise operators (binary) 287
Bitwise operators (unary) 288
Logical operators (unary) 289
Relational operators (binary) 289
HIGH operator 290
HIGH_6_13 Operator 291
LOW operator 291
MAP_ADDR_6 Operator 292
PAGE operator 292
Force operator (unary) 293
Operator precedence 294
Expression 295
Absolute expression 295
Simple relocatable expression 296
Unary operation result 297
Binary operations result 298
Translation limits 298

11 Assembler Directives 299

Directive overview 299
- Section-Definition directives. 299
- Constant-Definition directives 299
- Data-Allocation directives. 300
- Symbol-Linkage directives 300
- Assembly-Control directives. 301
- Listing-File Control directives 302
- Macro Control directives. 303
- Conditional Assembly directives 303

Detailed descriptions of all assembler directives 304
- ABSENTRY - Application entry point 304
- ALIGN - Align Location Counter. 305
- BASE - Set number base. 306
- CLIST - List conditional assembly 307
- DC - Define Constant 309
- DCB - Define Constant Block. 310
- DS - Define Space. 311
- ELSE - Conditional assembly 313
- END - End assembly. 314
- ENDFOR - End of FOR block 315
- ENDIF - End conditional assembly 316
- ENDM - End macro definition 317
- EQU - Equate symbol value 318
- EVEN - Force word alignment 319
- FAIL - Generate Error message. 320
- FOR - Repeat assembly block. 323
- IF - Conditional assembly 324
- IFcc - Conditional assembly 325
- INCLUDE - Include text from another file. 327
- LIST - Enable Listing 328
- LLEN - Set Line Length 329
- LONGEVEN - Forcing Long-Word alignment. 330
- MACRO - Begin macro definition 331

MEXIT - Terminate Macro Expansion . . . 332
MLIST - List macro expansions . . . 334
NOLIST - Disable Listing. . . . 336
NOPAGE - Disable Paging . . . 337
OFFSET - Create absolute symbols . . . 338
ORG - Set Location Counter. . . . 339
PAGE - Insert Page break . . . 340
PLEN - Set Page Length . . . 341
RAD50 - RAD50-encoded string constants . . . 342
SECTION - Declare Relocatable Section . . . 344
SET - Set Symbol Value . . . 346
SPC - Insert Blank Lines. . . . 347
TABS - Set Tab Length . . . 347
TITLE - Provide Listing Title . . . 347
XDEF - External Symbol Definition. . . . 348
XREF - External Symbol Reference . . . 349
XREFB - External Reference for Symbols located on the Direct Page . . . 349

12 Macros 351

Macro overview . . . 351
Defining a macro . . . 351
Calling macros . . . 352
Macro parameters . . . 352
Macro argument grouping. . . . 353
Labels inside macros. . . . 354
Macro expansion. . . . 356
Nested macros. . . . 356

13 Assembler Listing File 357

Page header. . . . 358
Source listing . . . 358
Abs. . . . 358
Rel. . . . 359
Loc. . . . 360
Obj. code . . . 361

Source line. 362

14 Mixed C and Assembler Applications 363

Memory models 363
Parameter passing scheme 364
Return Value 364
Accessing assembly variables in an ANSI-C source file 365
Accessing ANSI-C variables in an assembly source file 366
Invoking an assembly function in an ANSI-C source file 367
Example of a C file 368
Support for structured types 369
Structured type definition 370
Types allowed for structured type fields 370
Variable definition. 371
Variable declaration. 372
Accessing a structured variable. 372
Structured type: Limitations 374

15 Make Applications 375

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

16 How to... 377

Working with absolute sections 377
Defining absolute sections in an assembly source file 377
Linking an application containing absolute sections. 379
Working with relocatable sections 380
Defining relocatable sections in a source file 380
Linking an application containing relocatable sections 381
Initializing 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 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 . . . 393
Using the direct addressing mode to access external symbols . . . 393
Using the direct addressing mode to access exported symbols. . . . 394
Defining symbols in the direct page . . . 394
Using the force operator . . . 395
Using SHORT sections . . . 395

II Appendices

18 Global Configuration File Entries 399

[Installation] Section. . . . 399
Path . . . 399
Group. . . . 400
[Options] Section . . . 400
DefaultDir . . . 400
[XXX_Assembler] Section . . . 401
SaveOnExit . . . 401
SaveAppearance . . . 401
SaveEditor . . . 402
SaveOptions . . . 402
RecentProject0, RecentProject1 . . . 402
[Editor] Section. . . . 403
Editor_Name . . . 403
Editor_Exe. . . . 403
Editor_Opts . . . 404
Example . . . 405

19 Local Configuration File Entries 407

[Editor] Section. . . . 407

Editor_Name 407
Editor_Exe. 408
Editor_Opts 408
[XXX_Assembler] Section 409
RecentCommandLineX, X= integer 409
CurrentCommandLine. 409
StatusbarEnabled 410
ToolbarEnabled 410
WindowPos 411
WindowFont 411
TipFilePos 412
ShowTipOfDay 412
Options 413
EditorType. 413
EditorCommandLine. 414
EditorDDEClientName 414
EditorDDETopicName 414
EditorDDEServiceName 415
Example 416

20 MASM Compatibility 417

Comment Line 417
Constants (Integers) 417
Operators. 418
Directives 418

21 MCUasm Compatibility 421

Labels 421
SET directive 422
Obsolete directives 422

Index 423

Using HC(S)08/RS08 Assembler

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

Highlights

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

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

Structure of this Document

This section has the following chapters:

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

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

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

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

4

Working with Assembler

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

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

In this chapter:

- Programming Overview
- Managing Assembly Language Project Using CodeWarrior IDE
- Writing your Assembly Source Files
- Analyzing Project Files
- Assembling Source Files
- Linking Application
- Directly Generating ABS File
- Assembler Build Properties Panels

Programming Overview

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

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

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

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

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

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

The typical configuration of an Assembler is its association with a Project Directory and an External Editor. The CodeWarrior software uses the project directory for storing the files it creates and coordinates the various tools integrated into the CodeWarrior suite. The Assembler is but one of these tools that the IDE coordinates for your projects. The tools used most frequently within the CodeWarrior IDE are its Editor, Compiler, Assembler, Linker, the Simulator/Debugger, and Processor Expert. Most of these *Build Tools* are located in the `<MCU>\prog` subfolder of the CodeWarrior installation directory. The others are directly integrated into *CodeWarrior Development Studio for Microcontrollers V10.x*.

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

Project Directory

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

There are three methods of designing a project.

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

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

The default location this folder is:
`C:\Program Files\Freescale\CW MCU v10.x\MCU\prog`

External Editor

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

Managing Assembly Language Project Using CodeWarrior IDE

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

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

A Fibonacci series is an easily visualized infinite mathematical series:

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

It is simple to calculate the next number in this series. The first calculated result is actually the third number in the series because the first two numbers make up the starting point: 0

and 1. The next term in a Fibonacci series is the sum of the preceding two terms. The first sum is then: 0 + 1 = 1. The second sum is 1 + 1 = 2. The sixth sum is 5 + 8 = 13. And so on to infinity.

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

Create New Project

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

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

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

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

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

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

   The **Devices** page appears.

6. Expand the tree control and select the derivative or board you would like to use. For example, select **HCS08 > HCS08G Family > MC9S08GT60**.
7. Click **Next**.

   The **Connections** page appears.

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

   The **Languages** page appears.

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

    The **Rapid Application Development** page appears.

12. Select the appropriate rapid application development tool.

13. Click **Next**.

    The **Add Files** page appears.

14. If you want to add any file to your project, click . The **Add file path** dialog box appears. Specify the path of the file in the **File** text box and click **Finish**.

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

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

Select the project in the **CodeWarrior Projects** view. From the IDE menu bar, select **Project > Build Project** to build the project. The **Console** view displays the statements that direct the build tools to compile and link the project. The Binaries link appears, and so does the *<CPU Derivative>* folder (Figure 4.2).

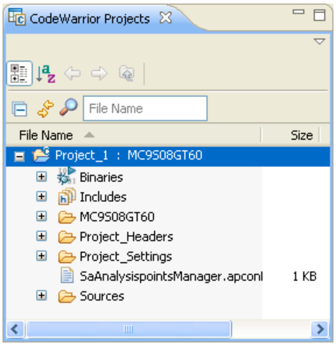
NOTE You can configure the IDE to build the project automatically. To configure the IDE to build the project automatically, check the **Build automatically** checkbox in the **Window > Preferences > General > Workspace** page.

Additional Project Information

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

The major GUI component for your project is the **CodeWarrior Projects** view, as when the project is created, the project appears in the **CodeWarrior Projects** view (Figure 4.1) in the Workbench window.

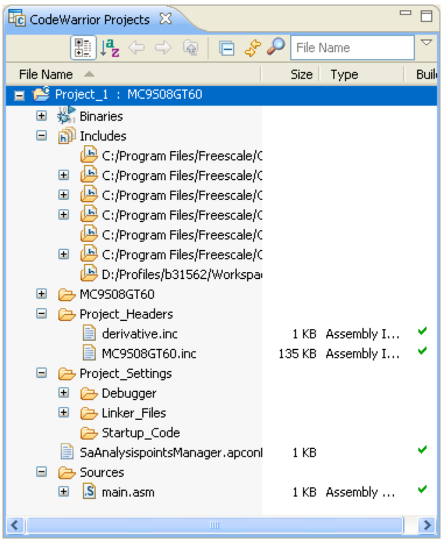
Figure 4.1 CodeWarrior Projects View



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

If you expand the folder icons, actually groups of files, by clicking in the **CodeWarrior Project** view, you can view the files created by the New Bareboard Project wizard (Figure 4.2).

Figure 4.2 CodeWarrior Projects View Showing Files



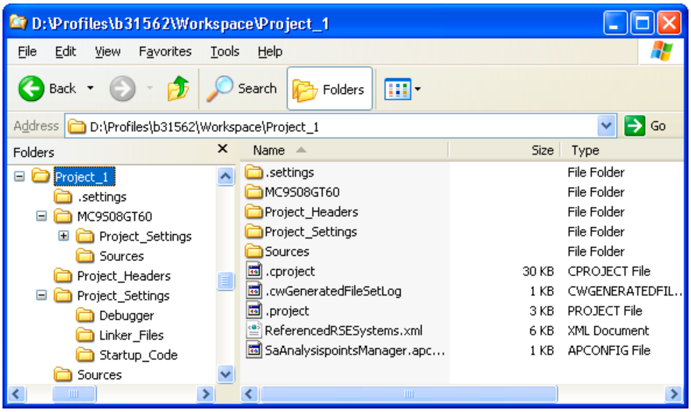
The expanded view displays the logical arrangement of the project files. At this stage, you can safely close the project and reopen it later, if desired.

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

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

Examine the project folder that the IDE generated when you created the project. To do this, right-click on the project's name (`Project_1 : MC9S08GT60`) in the CodeWarrior Projects view, and select **Show in Windows Explorer**. Windows displays the Eclipse workspace folder, along with the project folder, `Project_1`, within it (Figure 4.3).

Figure 4.3 Project Directory in Windows Explorer



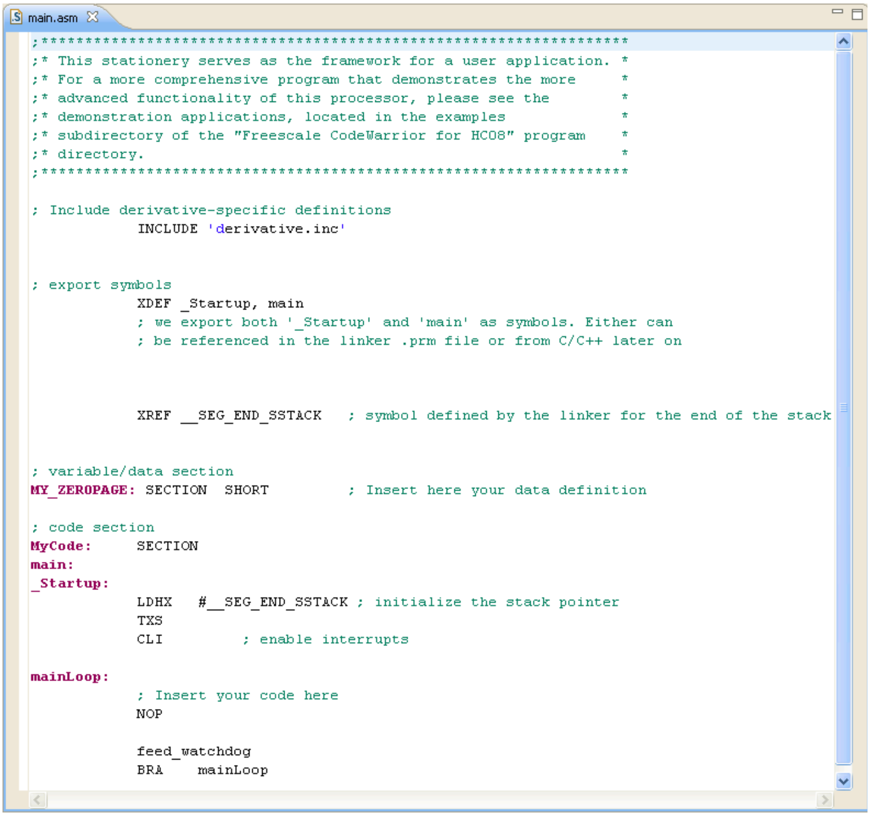
These are the actual folders and files generated for your project. When working with standalone tools, you may need to specify the paths to these files, so it is best that you know their locations and functions.

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

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

Double-click the `main.asm` file in the **Sources** group. The `main.asm` file opens in the editor area (Figure 4.4).

Figure 4.4 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 the "Freescale CodeWarrior for HC08" program    *
;* directory.                                                      *
;*******************************************************************

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


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



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


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

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

mainLoop:
            ; Insert your code here
            NOP

            feed_watchdog
            BRA    mainLoop
```

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

Analysis of Groups and Files in a Project

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

CodeWarrior Groups

These groups and their usual functions are:

- Sources

  This group contains the assembly source code files.

- Includes

  This group holds the C/C++ source files and the C header files.

- Project_Settings

  - Debugger

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

  - Linker Files

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

  - Startup Code

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

- Project_Headers

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

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

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

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

Creating New Group

To create a new group:

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

   The **New** dialog box appears.

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

   The **Group** wizard appears.

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

Adding New File to the Project

To add a new file to the project:

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

   The **New** dialog box appears.

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

   The **File** page appears.

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

Renaming File or Group

To rename a file or group:

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

Moving File

To move a file to a different group or project:

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

Removing File

To remove a file from a project:

1. Right-click the file in the **CodeWarrior Projects** view and select **Delete** from the context menu. A dialog box appears asking you to confirm the deletion.
2. Click **Yes**. This deletes the selected file from the project directory.

Restoring Deleted File

To restore a deleted file:

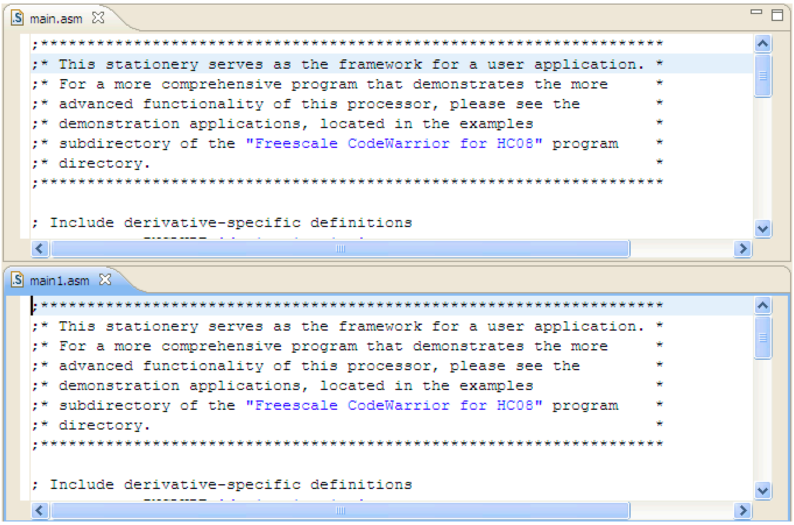
1. In the **CodeWarrior Projects** view, right-click the project to which the deleted file belongs.
2. Select **Restore from Local History** from the context menu. The **Restore from Local History** dialog box appears. The dialog box lists the deleted files available in the local history.
3. Check the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

Using Editor

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

button. By dragging the editor tabs, you can tile the source files in the editor area in order to view source files side by side (Figure 4.5).

Figure 4.5 Editor Area Showing Tiled Source Files



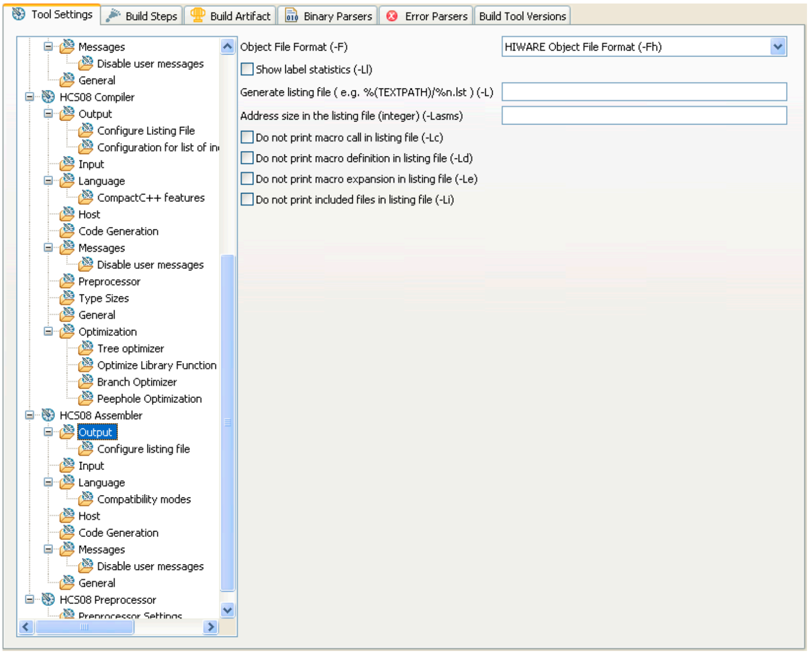
4. Drag the borders of the editor area or each editor, to resize as desired.
5. Make desired changes in the source file. To save the file perform any of the following:
   - Select **File > Save** from the IDE menu bar.
   - Right-click the file and select **Save** from the context menu.
   - Press **CTRL+S**.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

Generating Listing Files

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

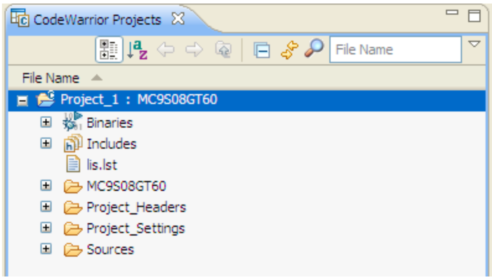
1. In the **CodeWarrior Projects** view, right-click the assembler project for which you want to generate output listing files.
2. Select **Properties** from the context menu that appears. The **Properties for *<project name>*** dialog box appears.
3. Select **C/C++ Build > Settings**. The **Tool Settings**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

Figure 4.6 HCS08 Assembler > Output Page



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

Figure 4.7 CodeWarrior Projects View — Assembly Listing File



Writing your Assembly Source Files

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

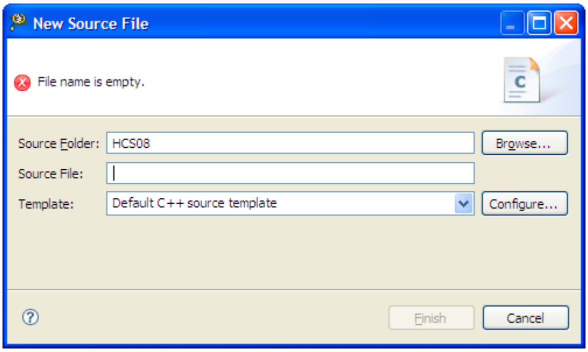
NOTE You can write an assembly application using one or several assembly units. Each assembly unit performs one particular task. An assembly unit is comprised of an assembly source file and, perhaps, some additional include files. Variables are exported from or imported to the different assembly units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

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

To create a new assembly source file:

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

Figure 4.8 New Source File Dialog Box



2. Enter the folder in which you want to add the new file in the **Source Folder** text box or click **Browse** to select the desired folder, for example `Project_2/Sources`.
3. Enter the name of the new file with extension `*.asm` in the **Source File** text box.
4. Click **Finish**. A newly created file opens in the editor area.
5. Write your assembly source code in the file.

Analyzing Project Files

We will analyze the default `main.asm` file that was generated when the project was created with the New Bareboard Project wizard (Listing 4.1). This is the assembler source code for the Fibonacci program.

Listing 4.1 main.asm file

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

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

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

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


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

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

mainLoop:
            ; Insert your code here
            NOP

            feed_watchdog
            BRA    mainLoop
```

Since the RS08 memory map is different from the HC08 memory map (and so is the instruction set), Listing 4.2 shows a similar example for RS08.

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

Listing 4.2 Contents of Source File for RS08 Derivative

```
;*******************************************************************
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more     *
;* advanced functionality of this processor, please see the        *
```

```
;* demonstration applications, located in the examples          *
;* subdirectory of the "Freescale CodeWarrior for HC08" program    *
;* directory.                                                      *
;*******************************************************************

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

; Include derivative-specific definitions
            INCLUDE 'derivative.inc'
;$$IF CLI                                            ; enable
interrupts
;$$// we should include here MCUInit.inc. Unfortunately, the one that
Unis generates does not assemble -> fix this when the fixed it.
;$$//; Include device initialization code
;$$//           INCLUDE 'MCUInit.inc'
;            XREF MCU_init
;$$ENDIF


; variable/data section
TINY_RAM_VARS: SECTION  RS08_SHORT         ; Insert here your data
definition
Counter:    DS.B   1
FiboRes:    DS.B   1
tmpCounter: DS.B   1
tmp:        DS.B   1


; code section
MyCode:     SECTION
main:
_Startup:
;$$IF CLI                                            ; enable
interrupts
            ; Call generated Device Initialization function
;            JSR    MCU_init
;$$ENDIF
mainLoop:


            CLRA                     ; A contains counter
cntLoop:    INCA
            CBEQA  #14,mainLoop      ; larger values cause overflow.
```

```
            MOV     #HIGH_6_13(SRS),PAGESEL
            STA     MAP_ADDR_6(SRS)   ; feed the watchdog

            STA     Counter           ; update global.
            BSR     CalcFibo
            STA     FiboRes           ; store result
            LDA     Counter
            BRA     cntLoop           ; next round.

CalcFibo:  ; Function to calculate fibonacci numbers. Argument is in A.
            DBNZA   fiboDo            ; fiboDo
            INCA
            RTS
fiboDo:
            STA     tmpCounter        ; the counter
            CLRX                      ; second last = 0
            LDA     #$01              ; last = 1
FiboLoop:   STA     tmp               ; store last
            ADDX
            LDX     tmp
            DBNZ    tmpCounter,FiboLoop
FiboDone:
            RTS                       ; result in A
```

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

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

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

Assembling Source Files

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

Assembling and Linking with CodeWarrior IDE

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

- <*assembly_source_file*>`.dbg`

  This file contains symbolic debugging information.

- <*project_name*>`.abs`

  This is the final executable file.

- <*project_name*>`.map`

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

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

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

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

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

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

Assembling with Assembler

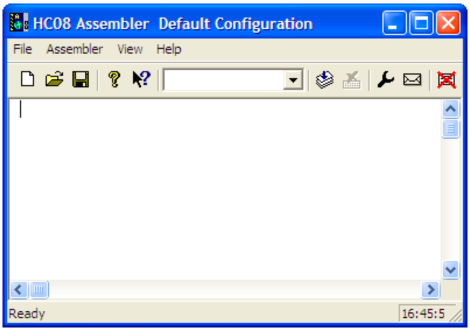
It is also possible to use the HC(S)08/RS08 Assembler as a standalone assembler. If you prefer not to use the assembler but you want to use the Linker, you can skip this section and proceed to Linking Application.

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

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

Start the Assembler by double-clicking the `ahc08.exe` file in the <*MCU*>`\prog` folder in the CodeWarrior installation directory. The Assembler opens (Figure 4.9). Read the tip displayed in the **Tip of the Day** dialog box, if you want to, and then click **Close** to close the dialog box.

Figure 4.9 HC08 Assembler Default Configuration Dialog Box



Configuring Assembler

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

- Global

  This data is common to all Build Tools and projects. There may be common data for each Build Tool, such as listing the most recent projects, etc. All tools may store some global data into the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (for example, `C:\WINDOWS`). See Listing 4.3.

Listing 4.3 Typical locations for a global configuration file

```
\<CWInstallDir>\MCU\prog\mcutools.ini - #1 priority
```

```
C:\WINDOWS\mcutools.ini - used if there is no mcutools.ini file above
```

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

```
C:\Program Files\Freescale\CW MCU
V10.x\<MCU>\prog\mcutools.ini
```

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

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

- Local

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

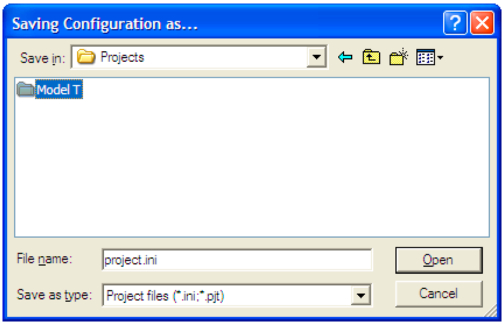
After opening the Assembler, you would load the configuration file for your project if it already had one. However, you will create a new configuration file for the project in this tutorial and save it so that when the project is reopened, its previously saved configuration state is used.

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

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

   The **Saving Configuration as** dialog box appears.
3. Navigate to the desired location and click the **Create New Folder** icon on the dialog box toolbar.
4. Enter a name for the project directory (Figure 4.10).

Figure 4.10 Loading Configuration Dialog Box



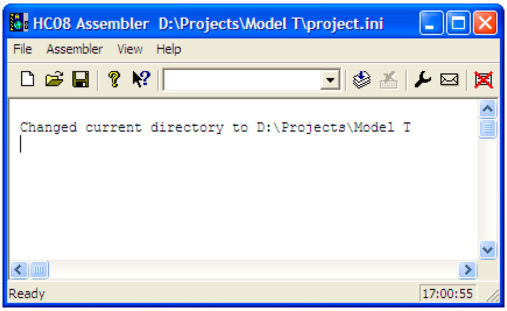
5. Click **Open**.

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

6. Click **Save**.

   The `project.ini` file is created in the `Model T` folder and becomes the local configuration file for this project. The current directory for the Microcontroller Assembler is changed to your project directory (Figure 4.11).

Figure 4.11 Assembler Displaying Current Directory



If you were to examine the project directory with the Windows Explorer at this point, it would only contain the `project.ini` configuration file that the Assembler just created (Figure 4.12).

Figure 4.12 Project directory in Windows Explorer

If you further examined the contents of the `project.ini` configuration file, you would see that it contains Assembler options in the `[AHC08_Assembler]` portion of the file. The `project.ini` file for this project only has an `[AHC08_Assembler]` section (Listing 4.4).

Listing 4.4 Contents of project.ini file

```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,371,209,798,496
EditorType=4
```

The `AHC08_Assembler` options are described in detail in [XXX_Assembler] Section in the Appendices.

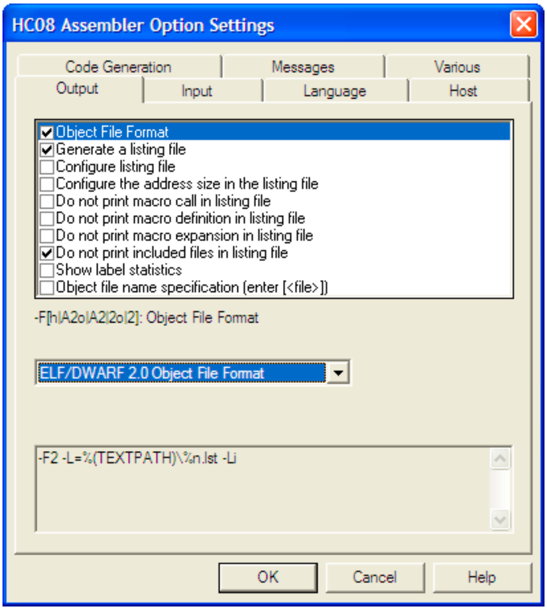
Next, you have to set the object-file format that you will use (HIWARE or ELF/DWARF).

1. Select **Assembler > Options**.

   The **HC08 Assembler Option Settings** dialog box (Figure 4.13) appears.

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

Figure 4.13 HC08 Assembler Option Settings Dialog Box



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

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

Save the changes to the configuration by:

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

After the changes to the configuration are saved, the `project.ini` file's contents are as follows (Listing 4.5).

Listing 4.5 project.ini file with additional assembly options

```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,308,151,767,337
EditorType=4
Options=-F2 -L=%(TEXTPATH)\%n.lst -Li
```

Input Files

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

For this project, you should have a project directory named `Model T`. Within this folder, you should have another folder named `Sources`, which contains the two files described above. Using a text editor of your choice, modify the `main.asm` file so that it appears as below (Listing 4.6):

Listing 4.6 main.asm File

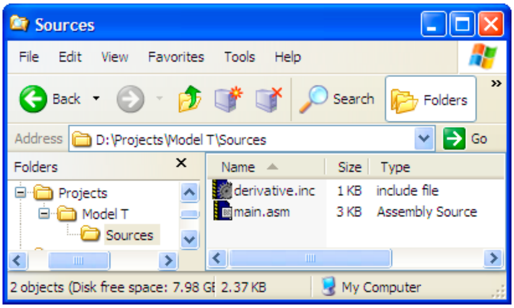
```
;*******************************************************************
;* This stationery serves as the framework for a user application.   *
;* For a more comprehensive program that demonstrates the more       *
;* advanced functionality of this processor, please see the          *
;* demonstration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

; export symbols
            XDEF _Startup, main
            ; we use export '_Startup' as symbol. This allows us to
            ; reference '_Startup' either in the linker .prm file
            ; or from C/C++ later on

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

      ; Include derivative-specific definitions

      INCLUDE 'derivative.inc'

; variable/data section
MY_ZEROPAGE: SECTION  SHORT         ; Insert here your data definition
Counter:    DS.B   1
FiboRes:    DS.B   1


; code section
MyCode:     SECTION
main:
_Startup:
            LDHX  #__SEG_END_SSTACK ; initialize the stack pointer
```

```
             TXS
             CLI                       ; enable interrupts
mainLoop:
             CLRA                      ; A contains counter
cntLoop:     INCA
             CBEQA  #14,mainLoop       ; larger values cause overflow.

             STA    Counter            ; update global.
             BSR    CalcFibo
             STA    FiboRes            ; store result
             LDA    Counter
             BRA    cntLoop            ; next round.
; Function to calculate fibonacci numbers. Argument is in A.
CalcFibo:
             DBNZA fiboDo              ; fiboDo
             INCA
             RTS
fiboDo:
             PSHA                      ; the counter
             CLRX                      ; second last = 0
             LDA    #$01               ; last = 1
FiboLoop:    PSHA

             ADD    1,SP
             PULX
             DBNZ   1,SP,FiboLoop
FiboDone:    PULH                   ; release counter
             RTS                    ; result in A
```

Now there are three files in the project (Figure 4.14):

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

Figure 4.14 Project Files

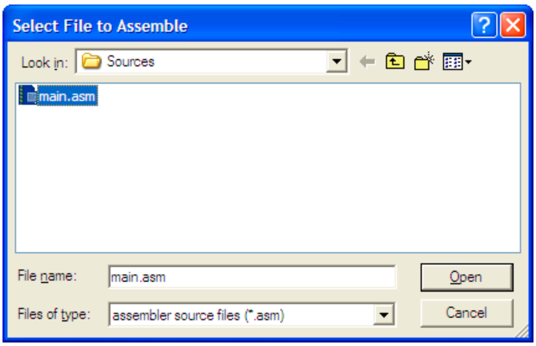


Assembling Assembly Source-code Files

Let’s assemble the `main.asm` file.

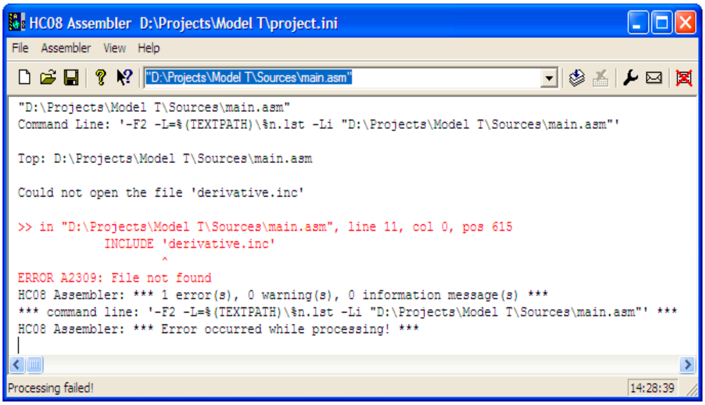
1. Select **File** > **Assemble** from the menu bar.
2. The **Select File to Assemble** dialog box appears (Figure 4.15).Browse to the `Sources` folder in the project directory and select the `main.asm` file.

Figure 4.15 Select File to Assemble Dialog Box



3. Click **Open** and the `main.asm` file should start assembling (Figure 4.16).

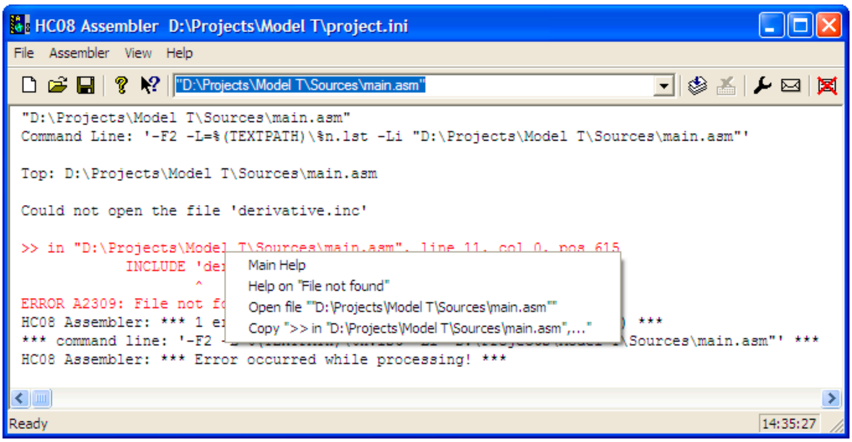
Figure 4.16 Results of Assembling main.asm File



The project window provides information about the assembly process or generates error messages if the assembly was unsuccessful. In this case, the *A2309 File not found* error message is generated. If you right-click on the text containing the error message, a context menu appears (Figure 4.17).

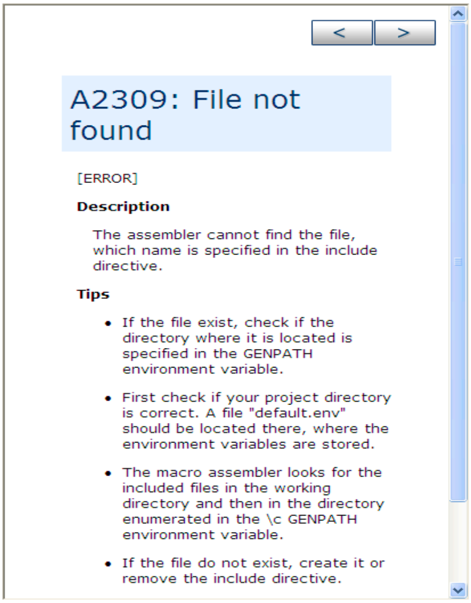
NOTE If you get any other types of errors, make sure the `main.asm` file is modified as shown in Listing 4.6

Figure 4.17 Context Menu



Select **Help on "file not found"** and help for the `A2309` error message appears (Figure 4.18).

Figure 4.18 A2309: File not found



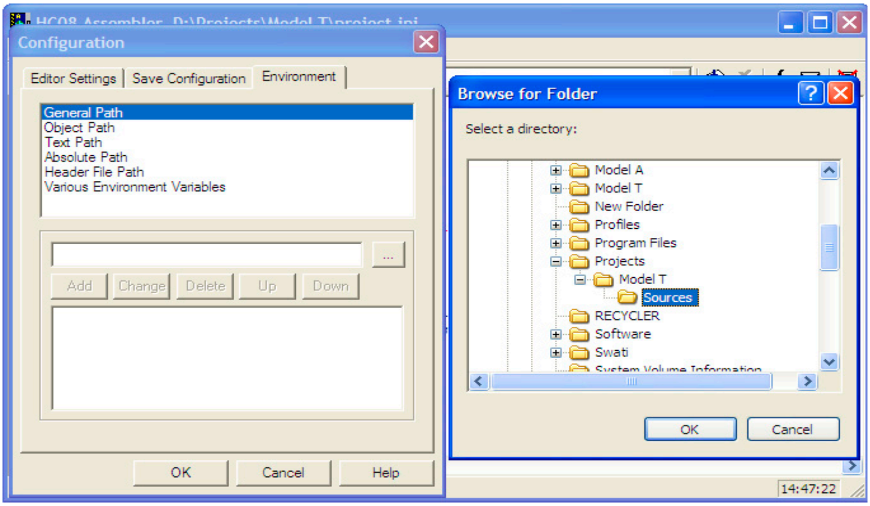
You know that the file exists because it is included in the `Sources` folder that you imported into the project directory. The help message for the `A2309` error states that the Assembler is looking for this "missing" include file first in the current directory and then in the directory specified by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

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

To fix this error:

1. Select **File > Configuration**.
2. The **Configuration** dialog box appears (Figure 4.19). Click the **Environment** tab and then select **General Path**.
3. Press the "..." button and navigate in the **Browse for Folder** dialog box for the folder that contains the `derivative.inc` file – the `Sources` folder in the project directory.

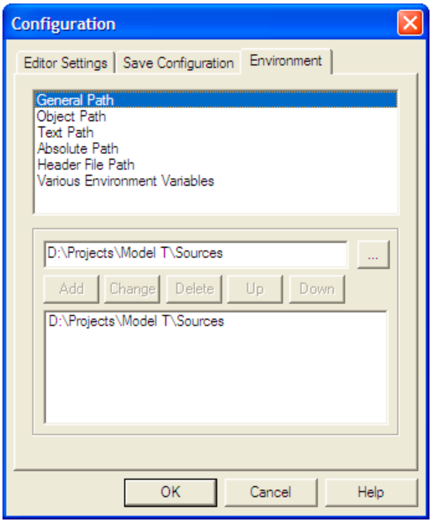
Figure 4.19 Browsing for Sources Folder



4. Click **OK** to close the **Browse for Folder** dialog box.
5. The **Configuration** dialog box is active again (Figure 4.20). Click the **Add** button

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

Figure 4.20 Adding GENPATH



6. Click **OK**.

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

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

   The asterisk disappears.

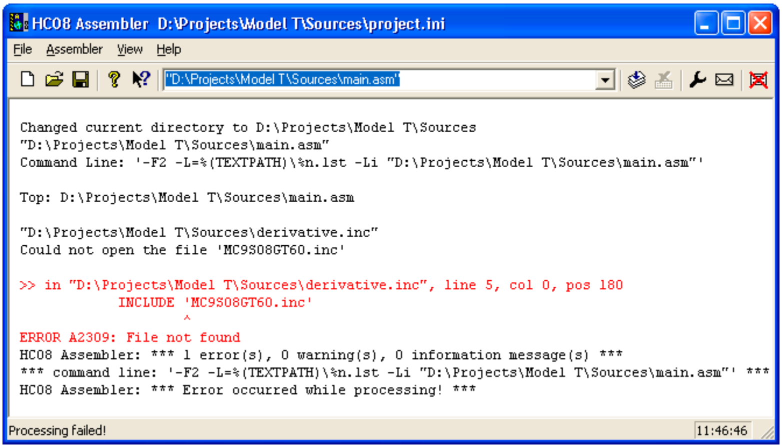
TIP You can clear the messages in the Assembler window at any time by selecting **View > Log > Clear Log**.

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 click **Open**. However, the `A2309` error message reappears but this time for a different include file - `mc9s08ac128.inc`. (Figure 4.21).

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

Figure 4.21 Assemble Attempt #2



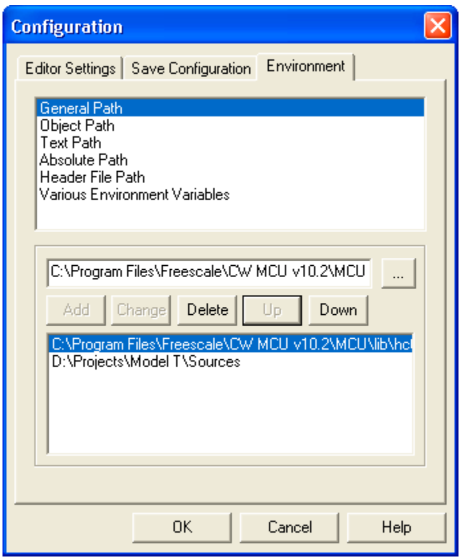
Fix this by repeating the `GENPATH` routine for the other include file (Figure 4.22). The `mc9s08ac128.inc` file is located at this path:

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

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

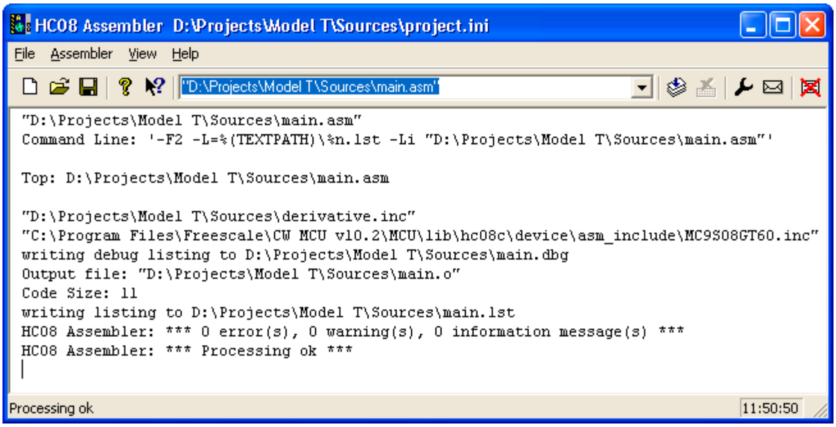
The `asm_include` folder is the typical place for missing include files.

Figure 4.22 Adding Another GENPATH



After the `GENPATH` is set up for the second include file and saved as before, you can try to assemble the `main.asm` file for the third time (Figure 4.23).

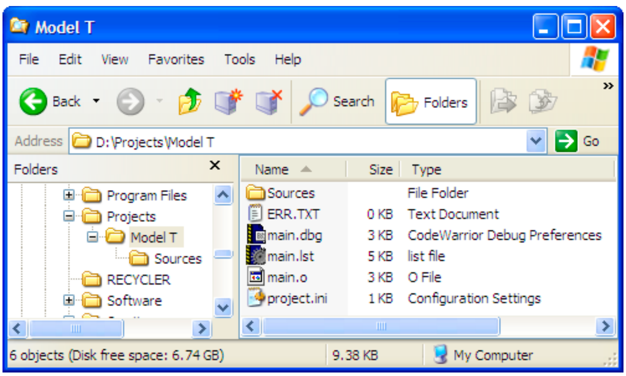
Figure 4.23 Assemble Attempt #3 - Success!



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

The Assembler also generated a `main.dbg` file (for use with the Simulator/Debugger), a `main.o` object file (for further processing with the Linker), and a `main.lst` output listing file in the project directory. The binary object-code file has the same name as the input module, but with the `*.o` extension – `main.o`. The debug file has the same name as the input module, but with the `*.dbg` extension – `main.dbg` and the assembly output listing file has the `*.lst` extension (Figure 4.24).

Figure 4.24 Project Directory After Successful Assembly



The ERR.TXT file is present in the project directory because of the earlier failed attempts at assembling. The ERR.TXT file is empty after a successful assembly. You can delete this file. Let's take an additional look at the project.ini file (Listing 4.7).

Listing 4.7 project.ini file after GENPATH environment variable is created

```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,171,56,854,423
EditorType=4
Options=-F2 -L=%(TEXTPATH)\%n.lst -Li
CurrentCommandLine=""D:\Projects\Model T\Sources\main.asm""
RecentCommandLine0=""D:\Projects\Model T\Sources\main.asm""
RecentCommandLine1=D:\Workspace\test\Sources\main.asm
[Environment Variables]
GENPATH=C:\Program Files\Freescale\CW MCU
V10.0\MCU\lib\hc08c\include;D:\Projects\Model T\Sources
OBJPATH=
TEXTPATH=
ABSPATH=
LIBPATH=
```

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

Linking Application

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

Linking with CodeWarrior IDE

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

PRM File

If you are using the CodeWarrior IDE to manage your project, a pre-configured PRM file for a particular derivative is already set up (Listing 4.8). Listing 4.9 is an example Linker PRM file for the RS08 derivative.

Listing 4.8 Linker PRM file for mc9s08gt60 derivative – Project.prm

```
/* This is a linker parameter file for the mc9s08gt60 */

NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */
    Z_RAM                    =  READ_WRITE   0x0080 TO 0x00FF;
    RAM                      =  READ_WRITE   0x0100 TO 0x107F;
    ROM                      =  READ_ONLY    0x182C TO 0xFFAF;
    ROM1                     =  READ_ONLY    0x1080 TO 0x17FF;
    ROM2                     =  READ_ONLY    0xFFC0 TO 0xFFCB;
 /* INTVECTS                =  READ_ONLY    0xFFCC TO 0xFFFF; Reserved
for Interrupt Vectors */
END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */
    DEFAULT_RAM,                        /* non-zero page variables */
                                        INTO  RAM;

    _PRESTART,                          /* startup code */
    STARTUP,                            /* startup data structures */
    ROM_VAR,                            /* constant variables */
    STRINGS,                            /* string literals */
    VIRTUAL_TABLE_SEGMENT,             /* C++ virtual table segment */
    DEFAULT_ROM,
    COPY                                /* copy down information: how
to initialize variables */
                                      INTO  ROM; /* ,ROM1,ROM2: To use
"ROM1,ROM2" as well, pass the option -OnB=b to the compiler */

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

STACKSIZE 0x50
```

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

Listing 4.9 Linker PRM file for RS08 derivative – Project.prm

```
NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */
    TINY_RAM                 =  READ_WRITE   0x0005 TO 0x000D;
    RAM                      =  READ_WRITE   0x0020 TO 0x004F;
    RESERVED_RAM             =  NO_INIT      0x0000 TO 0x0004;
    ROM                      =  READ_ONLY    0x3C00 TO 0x3FF7;
END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */
    RESERVED                 INTO RESERVED_RAM;
    TINY_RAM_VARS            INTO TINY_RAM;
    DIRECT_RAM_VARS          INTO RAM, TINY_RAM;
    DEFAULT_RAM              INTO RAM, TINY_RAM;
    DEFAULT_ROM              INTO ROM;
END

STACKSIZE 0x00 /* no stack for RS08 */

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

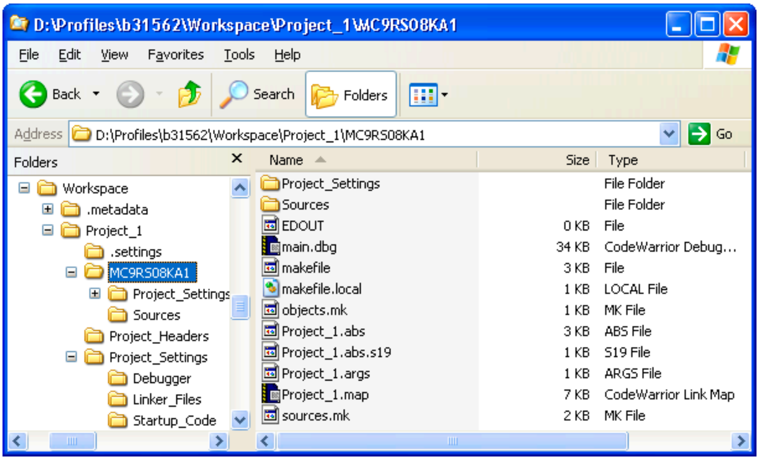
The Linker PRM file allocates memory for the stack and the sections named in the assembly source code files. If the sections in the source code are not specifically referenced in the `PLACEMENT` section, then these sections are included in `DEFAULT_ROM` or `DEFAULT_RAM`.

The `STACKSIZE` entry is used to set the stack size. The size of the stack for this project is 80 bytes. Some entries in the Linker PRM file may be commented-out by the IDE, as are the three last items in the `Project.prm` file in Listing 4.8.

Linking Object-code Files

You can build this relocatable assembly project by selecting **Project > Build Project** from the IDE menu bar. When the project is built, the Linker generates a `*.abs` and a `*.map` file in the <*CPU Derivative*> subfolder in the project directory (Figure 4.25).

Figure 4.25 Contents of <*CPU Derivative*> Subfolder After Building Project



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

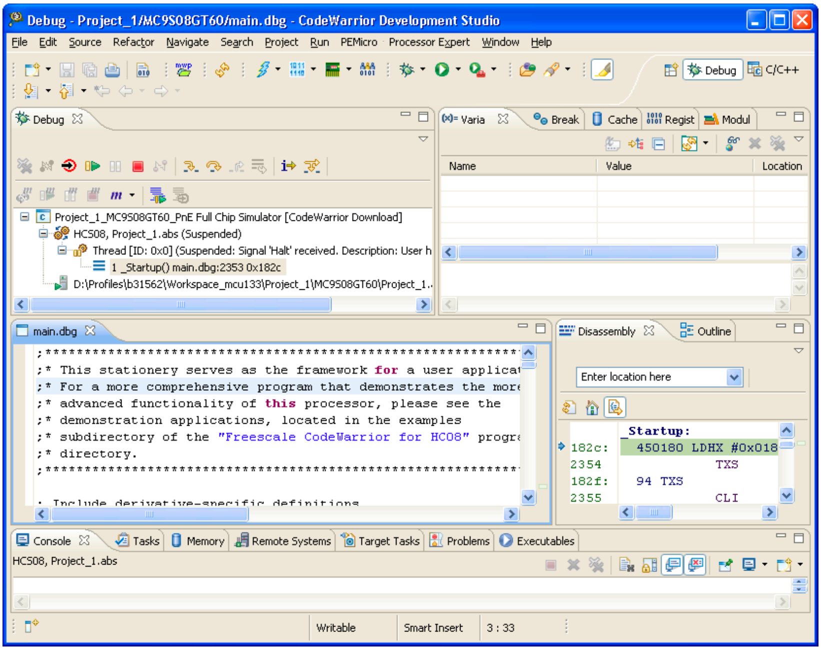
To debug the project:

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

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

2. Expand the **CodeWarrior Download** configuration.
3. From the expanded list, select the debug configuration that you want to modify.
4. Click the **Debugger** tab. The **Debugger** page appears in the area beneath the tabs.
5. Change the settings on this page as per your requirements. For example, select the required target processor and simulator/emulator.
6. Click the **Apply** button to save the new settings.
7. Click the **Debug** button to start the debugging session in the Debug perspective (Figure 4.26).

Figure 4.26 Debug Perspective



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

The Debug perspective displays information about:

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

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

Linking with Linker

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

1. Using a text editor, create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as `<project_name>.prm`.
2. Store the PRM file in a convenient location, such as the project directory.

3. In the <project_name>.prm file, change the name of the executable (*.abs) file to whatever you choose, for example, <project_name>.abs. In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's Model T.prm file (a PRM file for MC9S08GT60 from another CodeWarrior project was adapted) is shown in Listing 4.10.

Listing 4.10 Layout of a PRM file for the Linker – Model T.prm

```
/* This is a linker parameter file for the mc9s08gt60 */

LINK Model_T.abs /* Absolute executable file */
NAMES main.o /* Input object-code files are listed here. */
END

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
PLACEMENT below. */
    Z_RAM                    =  READ_WRITE   0x0080 TO 0x00FF;
    RAM                      =  READ_WRITE   0x0100 TO 0x107F;
    ROM                      =  READ_ONLY    0x182C TO 0xFFAF;
    ROM1                     =  READ_ONLY    0x1080 TO 0x17FF;
    ROM2                     =  READ_ONLY    0xFFC0 TO 0xFFCB;
 /* INTVECTS                =  READ_ONLY    0xFFCC TO 0xFFFF; Reserved
for Interrupt Vectors */
END

PLACEMENT /* Here all predefined and user segments are placed into the
SEGMENTS defined above. */
    DEFAULT_RAM,                        /* non-zero page variables */
                                        INTO  RAM;

    _PRESTART,                          /* startup code */
    STARTUP,                            /* startup data structures */
    ROM_VAR,                            /* constant variables */
    STRINGS,                            /* string literals */
    VIRTUAL_TABLE_SEGMENT,             /* C++ virtual table segment */
    DEFAULT_ROM,
    COPY                                /* copy down information: how
to initialize variables */
                                     INTO  ROM; /* ,ROM1,ROM2: To use
"ROM1,ROM2" as well, pass the option -OnB=b to the compiler */

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

STACKSIZE 0x50
```

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

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

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

```
MyCode:   SECTION   ; code section
main:
_Startup:
   LDHX #__SEG_END_SSTACK ; initialize stack pointer
   TXS
```

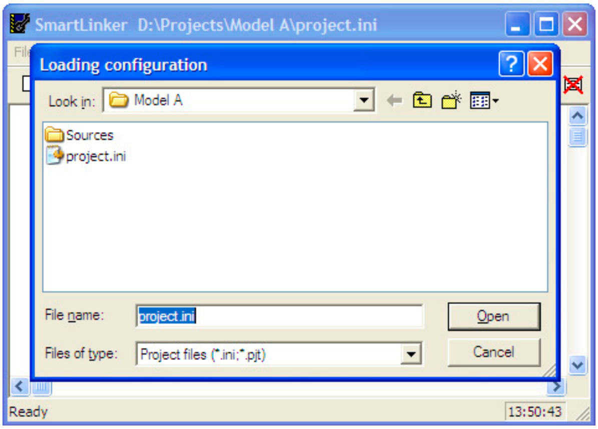
The statements in the linker parameter file are described in the Linker portion of the Build Tool Utilities manual.

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

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

   Select **File > Load Configuration** and navigate to and select the project's configuration file (Figure 4.27).

Figure 4.27 Microcontroller Linker



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

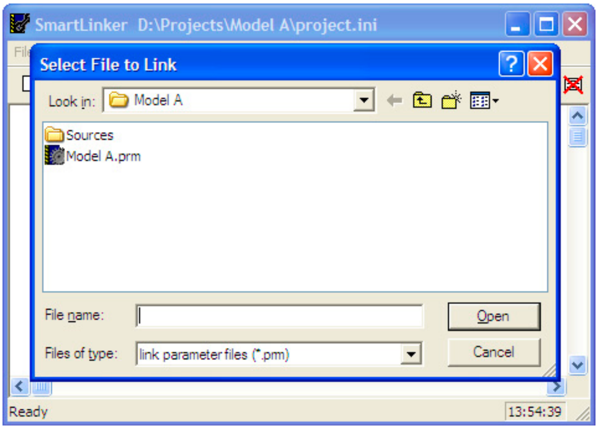
   The project directory is now the current directory for the Linker.

8. Select **File** > **Save Configuration** to save the configuration.

9. Select **File > Link**. The **Select File to Link** dialog box appears (Figure 4.28).

10. Browse to locate and select the PRM file for your project.

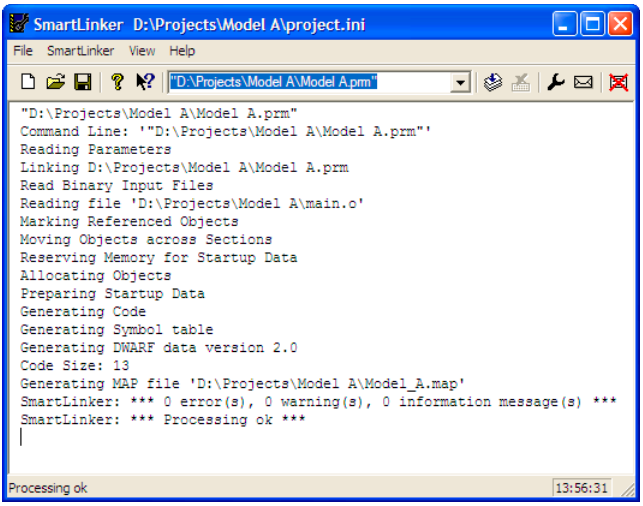
Figure 4.28 Select File to Link Dialog Box



11. Click **Open**.

    The Smart Linker links the object-code files in the `NAMES` section to produce the executable `*.abs` file, as specified in the `LINK` portion of the Linker PRM file (Figure 4.29).

Figure 4.29 SmartLinker Window After Linking

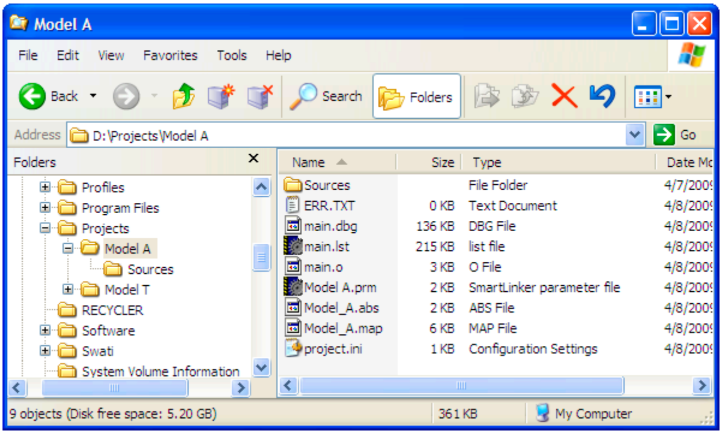


The messages in the linker's project window indicate that:

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

The `TEXTPATH` environmental variable was not used for this project. Therefore, the Linker generates its `*.map` Linker Map file in the same folder that contains the PRM file for the project. Because the `ABSPATH` environment variable was not used, the `*.abs` executable file is generated in the same folder as the Linker PRM file. Figure 4.30 shows the contents of the project directory after the relocatable assembly project is linked.

Figure 4.30 Project Directory After Linking



Directly Generating ABS File

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

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

Creating Absolute Assembly Project

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

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

   The **New Bareboard Project** wizard launches – the **Create an MCU bareboard Project** page appears.
3. Specify a name for the new project. For example, enter the project name as `AbsoluteAssembly`.
4. Click **Next**.

   The **Devices** page appears.
5. Expand the tree control and select the derivative or board you would like to use.

6. Click **Next**.

   The **Connections** page appears.

7. Select the appropriate connection(s).

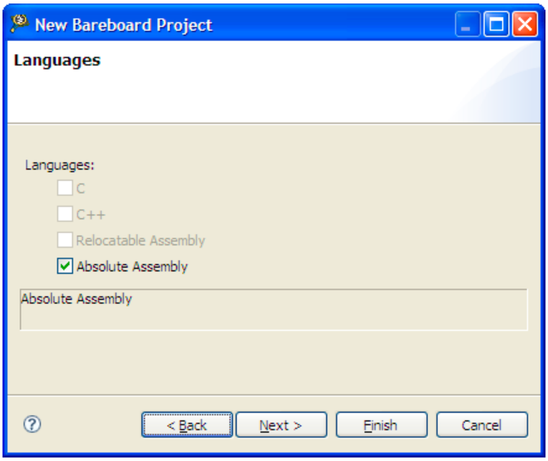
8. Click **Next**.

   The **Languages** page appears.

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

10. Check the **Absolute Assembly** checkbox.

Figure 4.31 Language Page



11. Click **Next**.

    The **Rapid Application Development** page appears.

12. Select the appropriate option.

13. Click **Next**.

14. The **Add Files** page appears.

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

Adapting Absolute Assembly File Created by Wizard

Modify the absolute assembly file `main.asm` as shown in Listing 4.11

The ORG directives must specify the absolute memory areas for ROM and RAM. Listing 4.11 is an adaptation of the main.asm file produced previously by the Wizard. This file may be used by the Assembler build tool or IDE to directly generate an ABS file.

Listing 4.11 Example Source File — main.asm

```
;********************************************************************
;* This stationery serves as the framework for a user              *
;* application. For a more comprehensive program that              *
;* demonstrates the more advanced functionality of this            *
;* processor, please see the demonstration applications            *
;* located in the examples subdirectory of CodeWarrior for         *
;* Microcontrollers V10.x program directory.                        *
;********************************************************************

; application entry point
             ABSENTRY  _Startup
; export symbols
            XDEF _Startup, main
            ; we use '_Startup' as an export symbol. This allows
            ; us to reference '_Startup' either in the linker
            ; *.prm file or from C/C++ later on.

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

; variable/data section
            ORG   $0040
Counter:    DS.B  1
FiboRes:    DS.B  1

; initial value for SP
initStack:  EQU   $023E

; code section
            ORG   $8000
main:
_Startup:
            LDHX  #initStack        ; initialize the stack pointer
            TXS
            CLI                     ; enable interrupts
mainLoop:
            CLRA                    ; A contains a counter.
cntLoop:    INCA
            CBEQA #14,mainLoop      ; Larger values cause overflow.
            feed_watchdog           ;
            STA   Counter           ; update global
            BSR   CalcFibo
```

```
            STA   FiboRes          ; store result
            LDA   Counter
            BRA   cntLoop          ; next round

CalcFibo: ; Function to compute Fibonacci numbers. Argument is in A.
            DBNZA fiboDo           ; fiboDo
            INCA
            RTS
fiboDo:
            PSHA                   ; the counter
            CLRX                   ; second last = 0
            LDA   #$01             ; last = 1
FiboLoop:   PSHA                   ; push last
            TXA
            ADD   1,SP
            PULX
            DBNZ  1,SP,FiboLoop
FiboDone:   PULH                   ; release counter
            RTS                    ; Result in A

;*******************************************************************
;* spurious - Spurious Interrupt Service Routine.                  *
;*               (unwanted interrupt)                              *
;*******************************************************************
spurious:                          ; Put here so the security
            NOP                    ; value does not change
            RTI                    ; all the time.

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

Listing 4.12 is a similar example for RS08.

Listing 4.12 Example Source File abstest_rs08.asm

```
ABSENTRY entry; Specifies the application Entry point
XDEF entry ; Make the symbol entry visible (needed for debugging)
        ORG $20 ; Define an absolute constant section
var1: DC.B 5 ; Assign 5 to the symbol var1
        ORG $40 ; Define an absolute data section
data: DS.B 1 ; Define one byte variable in RAM at $80
        ORG $3C00 ; Define an absolute code section
entry:
```

```
        LDA var1
main:
        INCA
        STA data
        BRA main
```

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

- The Reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application's entry point address is created at the reset vector address. To set the entry point of the application at address `$FFFA` on the `_Startup` label the following code is needed (Listing 4.13).

Listing 4.13 Setting the Reset vector address

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

The `ABSENTRY` directive is used to write the address of the application entry point in the generated absolute file. To set the entry point of the application on the `_Startup` label in the absolute file, the following code is needed (Listing 4.14).

Listing 4.14 Using ABSENTRY to enter the entry-point address

```
    ABSENTRY _Startup
```

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

Generating Absolute Assembly Using CodeWarrior IDE

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

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

The CodeWarrior IDE produces the same `*.abs` output files that the Assembler and Linker generated for relocatable assembly. The `*.sx` file generated in the `HCS08` folder of the project directory is a standard S-Record File. You can burn this file directly into a ROM memory.

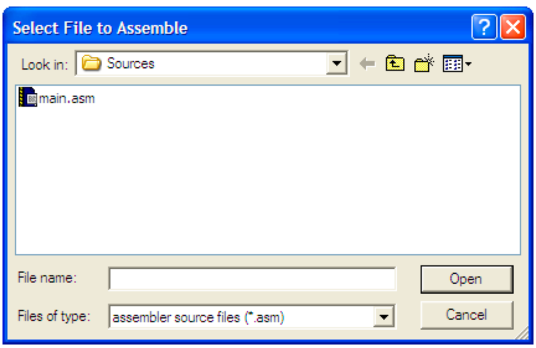
Generating Absolute Assembly Using Assembler Build Tool

Use the same project, `Model T`, that was used for the relocatable assembly project. Modify the `main.asm` in the `Model T\Sources` folder as per the Listing 4.11.

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

   The Assembler opens. Close the **Tip of the Day** dialog box.
2. Select **File > Load Configuration**. Browse for the project directory and set it as the current directory for the Assembler.
3. Select **Assembler > Options**. The **Option Settings** dialog box appears.
4. In the **Output** dialog box, check the **Object File Format** checkbox. A list box is displayed in the dialog box below the list of checkboxes.
5. Select the **ELF/DWARF 2.0 Absolute File** option from the list box. Click **OK**.
6. Select **File > Assemble**. The **Select File to Assemble** dialog box appears (Figure 4.32).

Figure 4.32 Select File to Assemble Dialog Box

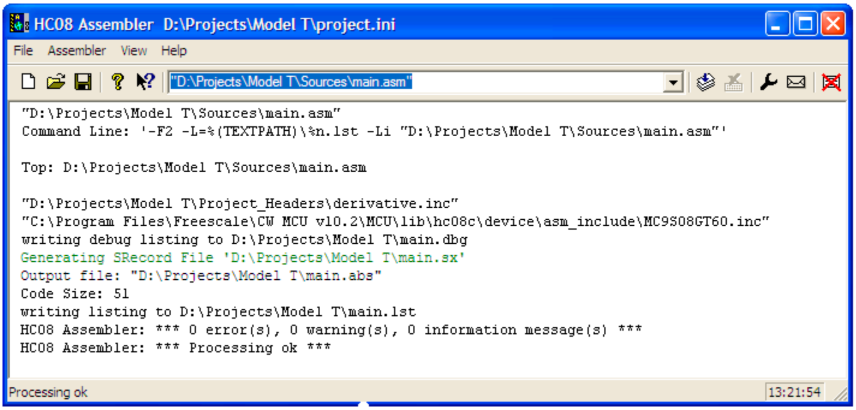


7. Browse and select the absolute-assembly source-code file `main.asm`.

8. Click **Open**.

   The Assembler now assembles the source code. Make sure that the `GENPATH` configurations are set for the two include files needed for the `main.asm` file in this project in case they have not yet been previously set. Messages about the assembly process are displayed in the assembler main window (Figure 4.33).

Figure 4.33 Successful Absolute Assembly



The messages indicate that:

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

Assembler Build Properties Panels

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

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

   The **Properties for *<project>*** dialog box appears.
3. Expand the **C/C++ Build** tree node and select **Settings**.

   The various settings for the build tools are displayed in the right panel. If not, click the **Tool Settings** tab. If you have selected an **HCS08** project, Figure 4.34 appears. If you have selected a **RS08** project, Figure 4.35 appears.

Figure 4.34 Project's Properties Dialog Box — HCS08 Project

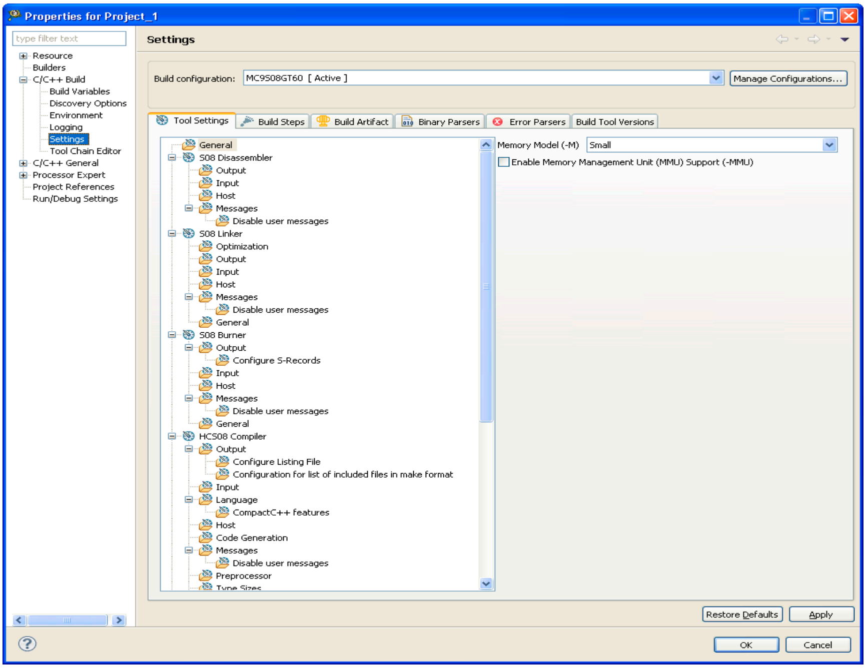
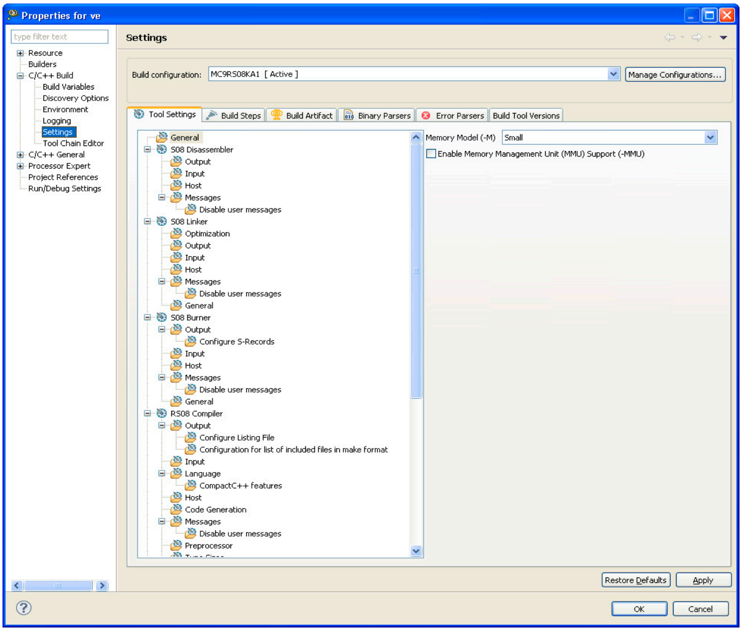


Figure 4.35 Project's Properties Dialog Box — RS08 Project



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

HCS08 Assembler Build Properties Panels

Table 4.1 lists the build properties panels for the HC(S)08 Assembler.

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

Table 4.1 Build Properties Panel for HC(S)08 Assembler

Build Tool	Build Properties Panels
HCS08 Assembler	HCS08 Assembler > Output HCS08 Assembler > Output > Configure listing file
	HCS08 Assembler > Input

Table 4.1 Build Properties Panel for HC(S)08 Assembler

Build Tool	Build Properties Panels
HCS08 Assembler	HCS08 Assembler > Language HCS08 Assembler > Language > Compatibility modes
	HCS08 Assembler > Host
	HCS08 Assembler > Code Generation
	HCS08 Assembler > Messages HCS08 Assembler > Messages > Disable user messages
	HCS08 Assembler > General

HCS08 Assembler

Use this panel to specify the command, options, and expert settings for the build tool assembler. Figure 4.36 shows the **Assembler** settings.

Figure 4.36 Tool Settings — Assembler

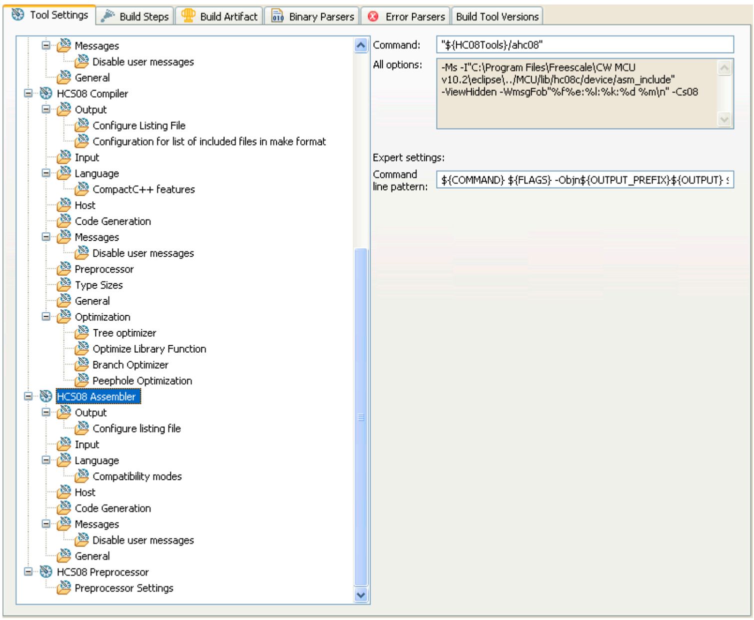


Table 4.2 lists and describes the assembler options for HCS08.

Table 4.2 Tool Settings — Assembler Options

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

HCS08 Assembler > Output

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

Figure 4.37 shows the **Output** panel.

Figure 4.37 Tool Settings — HCS08 Assembler > Output

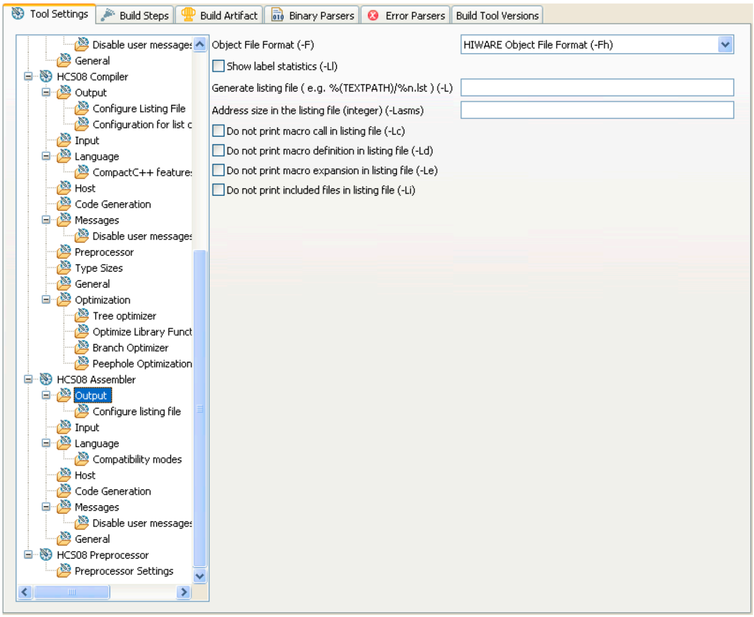


Table 4.3 lists and describes the output options for HCS08 Assembler.

Table 4.3 Tool Settings — HCS08 Assembler > Output Options

Option	Description
Object File Format (`-F`)	Defines the format for the output file generated by the Assembler.
Show label statistics (`-Ll`)	Enables the assembler to append statistical information about the compilation session to the specified file. The information includes assembler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) for each procedure of the compiled file. The assembler appends the information to the specified filename (or the file make.txt, if no argument given). Set the TEXTPATH: Text File Path environment variable to store the file into the path specified by the environment variable. Otherwise the assembler stores the file in the current directory.

Table 4.3 Tool Settings — HCS08 Assembler > Output Options (*continued*)

Option	Description
Generate listing file (for example, %(TEXTPATH)/ %n.lst) (`-L`)	Specifies the name, %n, of the assembly listing file. The file is placed in the directory specified by %TEXTPATH. If this option is left blank, no listing file is output.
Address size in the listing file (integer) (`-Lasms`)	Specifies the size of the addresses displayed in the listing. Options are: • 1 to display addresses as xx • 2 to display addresses as xxxx • 3 to display addresses as xxxxxx • 4 to display addresses asf xxxxxxxx
Do not print macro call in listing file (`-Lc`)	Specifies whether macro calls encountered in the source code are expanded and appear in the listing file.
Do not print macro definition in listing file (`-Ld`)	Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.
Do not print macro expansion in listing file (`-Le`)	Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.
Do not print included files in listing file (`-Li`)	Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

HCS08 Assembler > Output > Configure listing file

Use this panel to specify the general assembler behavior.

Figure 4.38 shows the **Configure listing file** panel.

Figure 4.38 Tool Settings — HCS08 Assembler > Output > Configure listing file

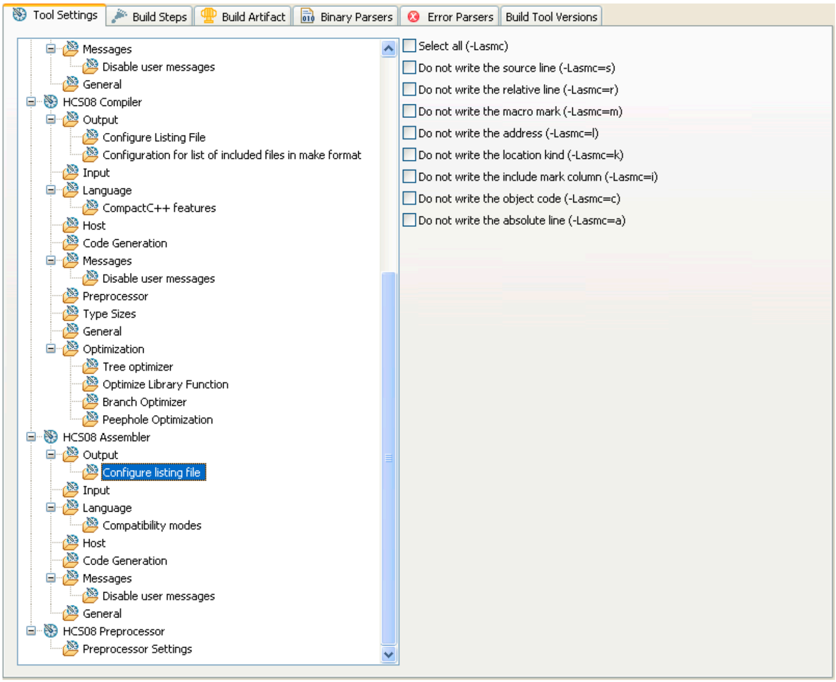


Table 4.4 lists and describes the configure listing file options for HCS08.

Table 4.4 Tool Settings — Assembler > Output > Configure listing file Options

Option	Description
Select all (`-Lasmc`)	Print all the columns in the listing file.
Do not write the source line (`-Lasmc=s`)	Do not print source column in the listing file.
Do not write the relative line (`-Lasmc=r`)	Do not print relative column (Rel.) in the listing file.
Do not write the macro line (`-Lasmc=m`)	Do not print macro mark column in the listing file.
Do not write the address (`-Lasmc=l`)	Do not print address column (Loc) in the listing file.

Table 4.4 Tool Settings — Assembler > Output > Configure listing file Options

Option	Description
Do not write the location kind (`-Lasmc=k`)	Do not print the location type column in the listing file.
Do not write the include mark column (`-Lasmc=i`)	Do not print the include mark column in the listing file.
Do not write the object code (`-Lasmc=c`)	Do not print the object code in the listing file.
Do not write the absolute line (`-Lasmc=a`)	Do not print the absolute column (Abs.) in the listing file.

HCS08 Assembler > Input

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

Figure 4.39 shows the **Input** panel.

Figure 4.39 Tool Settings — HCS08 Assembler > Input

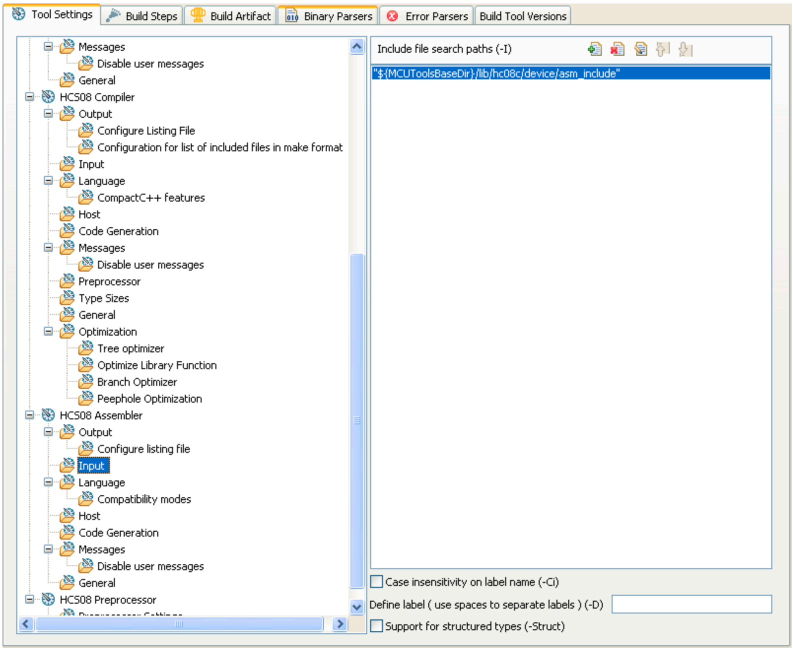


Table 4.5 lists and describes the input options for HCS08 Assembler.

Table 4.5 Tool Settings — HCS08 Assembler > Input Options

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

Table 4.6 lists and describes the toolbar buttons that help work with the file search paths.

Table 4.6 Include File Search Paths (-I) Toolbar Buttons

Button	Description
	Add — Click to open the Add directory path dialog box and specify the file search path.
	Delete — Click to delete the selected file search path.
	Edit — Click to open the Edit directory path dialog box and update the selected object file search path.
	Move up — Click to move the selected file search path one position higher in the list.
	Move down — Click to move the selected file search path one position lower in the list.

HCS08 Assembler > Language

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

Figure 4.40 shows the **Language** panel.

Figure 4.40 Tool Settings — HCS08 Assembler > Language

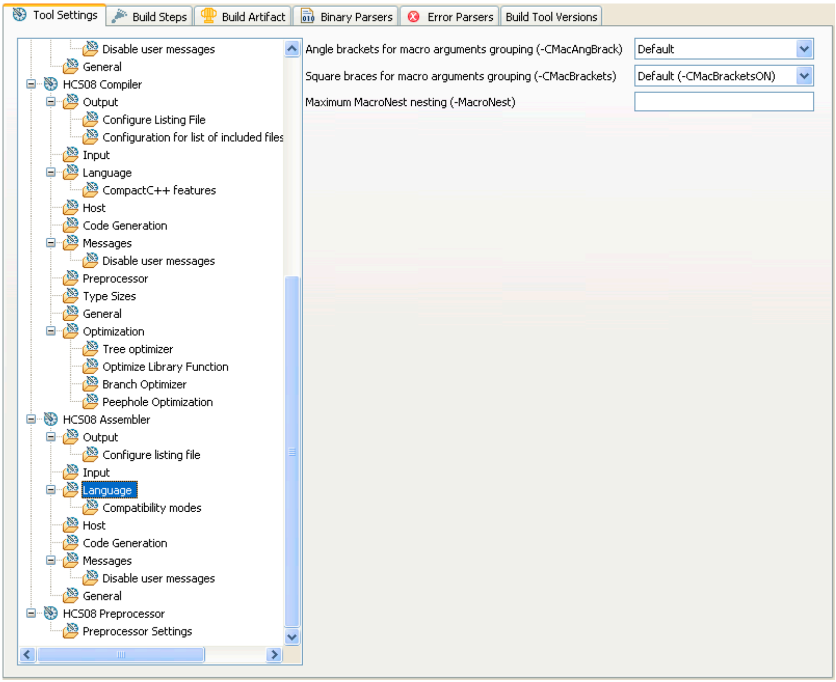


Table 4.7 lists and describes the language options for HCS08 Assembler.

Table 4.7 Tool Settings — HCS08 Assembler > Language Options

Option	Description
Angle brackets for macro arguments grouping (`-CMacAngBrack`)	Controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the [? ?] syntax instead. Options are: • Allow • Disallow
Square braces for macro arguments grouping (`-CMacBrackets`)	Controls the availability of the [? ?] syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for [? in the macro invocation context. Options are: • `-CMacBracketsON` • `-CMacBracketsOFF`
Maximum MacroNest nesting (`-MacroNest`)	Controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

HCS08 Assembler > Language > Compatibility modes

Figure 4.41 shows the **Compatibility modes** panel.

Figure 4.41 Tool Settings — HCS08 Assembler > Language > Compatibility modes

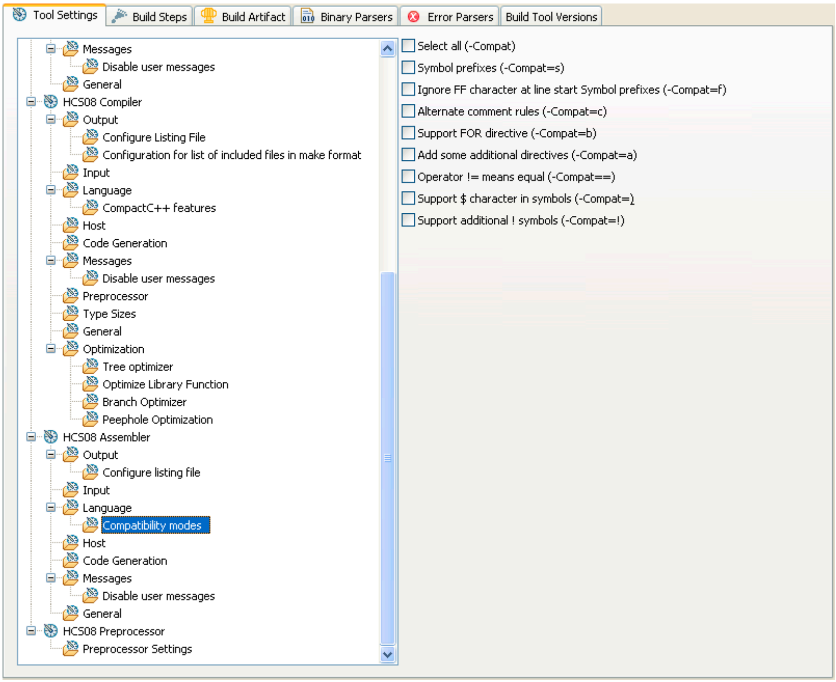


Table 4.8 lists and describes the compatibility mode options for HCS08 Assembler.

Table 4.8 Tool Settings — HCS08 Assembler > Language > Compatibility mode Options

Option	Description
Select all (`-Compat`)	Check to enable all compatibility mode options.
Symbol prefixes (`-Compat=s`)	With this suboption, the Assembler accepts "pgz:" and "byte:" prefixed for symbols in XDEFs and XREFs. They correspond to XREF.B or XDEF.B with the same symbols without the prefix.
Ignore FF character at line start Symbol prefixes (`-Compat=f`)	With this suboption, an otherwise improper character recognized from feed character is ignored.

Table 4.8 Tool Settings — HCS08 Assembler > Language > Compatibility mode Options

Option	Description
Alternate comment rules (`-Compat=c`)	With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler does issue a warning if such a comment does not start with a "*" or a ";".
Support FOR directive (`-Compat=b`)	With this suboption, the Assembler supports a FOR - Repeat assembly block assembly directive to generate repeated patterns more easily without having to use recursive macros.
Add some additional directives (`-Compat=a`)	With this suboption, some additional directives are added for enhanced compatibility. The Assembler actually supports a SECT directive as an alias of the usual SECTION - Declare Relocatable Section assembly directive. The SECT directive takes the section name as its first argument.
Operator != means equal (`-Compat==`)	The Assembler takes the default value of the != operator as not equal, as it is in the C language. For compatibility, this behavior can be changed to equal with this option. Because of the risks involved with this option for existing code, a message is issued for every != which is treated as equal.

Table 4.8 Tool Settings — HCS08 Assembler > Language > Compatibility mode Options

Option	Description
Support $ character in symbol (`-Compat=`)	With this suboption, the Assembler supports to start identifiers with a $ sign.
Support additional ! symbols (`-Compat=!`)	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

HCS08 Assembler > Host

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

Figure 4.42 shows the Host settings.

Figure 4.42 Tool Settings — HCS08 Assembler > Host

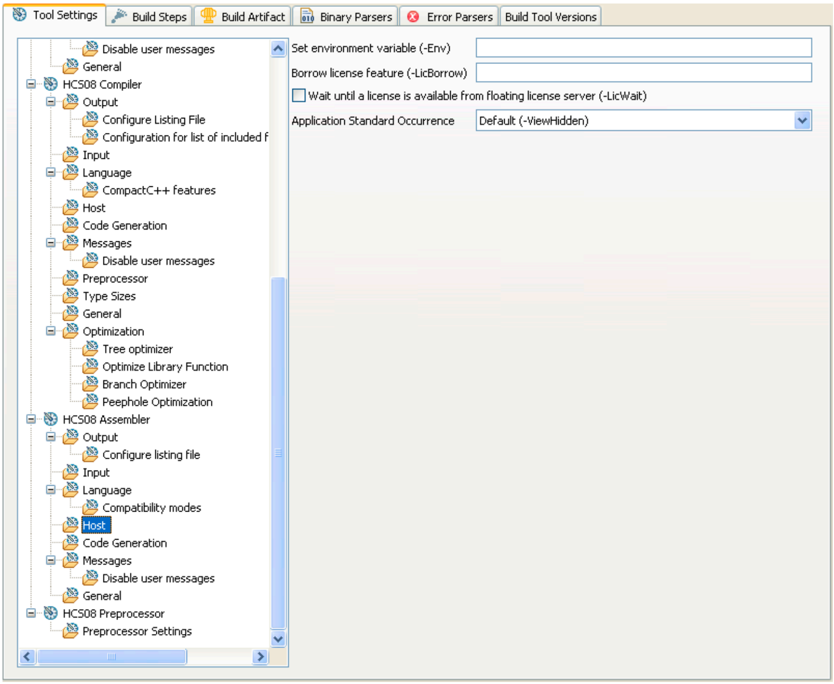


Table 4.9 lists and describes the memory model options for HCS08.

Table 4.9 Tool Settings — HCS08 Assembler > Host Options

Option	Description
Set environment variable (`-Env`)	This option sets an environment variable. Use this environment variable in the maker, or use to overwrite system environment variables.
Borrow license feature (`-LicBorrow`)	This option allows you to borrow a license feature until a given date or time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

Table 4.9 Tool Settings — HCS08 Assembler > Host Options (*continued*)

Option	Description
Wait until a license is available from floating license server (`-LicWait`)	By default, if a license is not available from the floating license server, then the application will immediately return. With `-LicWait` set, the application will wait (blocking) until a license is available from the floating license server.
Application Standard Occurrence	This option allows you to select the standard appearance for the application window. By default the option **-ViewHidden** is selected. For more Information, refer to the section “-View: Application standard occurrence” on page 212

HCS08 Assembler > Code Generation

Use this panel to specify the code generation assembler behavior.

Figure 4.43 shows the **Code Generation** panel.

Figure 4.43 Tool Settings — HCS08 Assembler > Code Generation

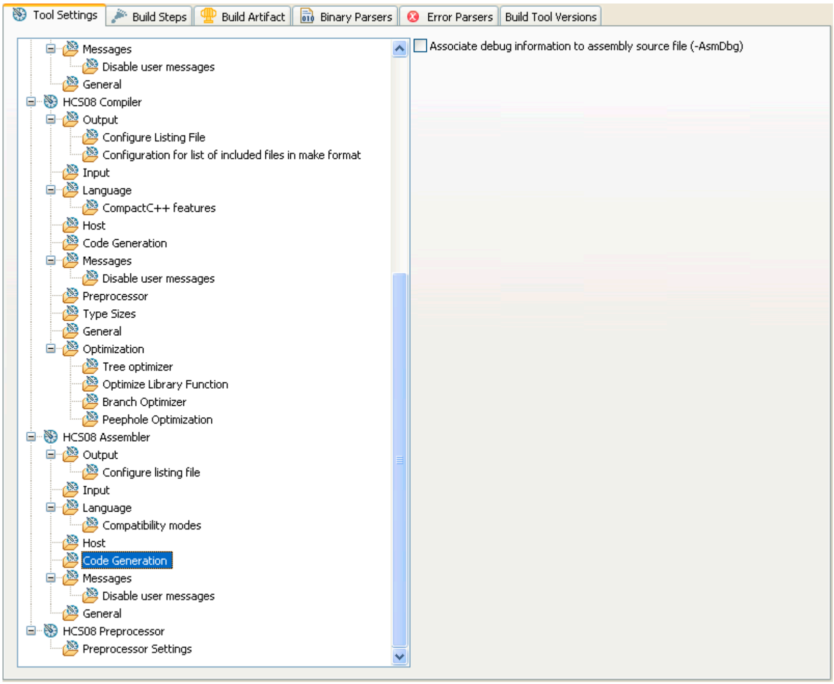


Table 4.10 lists and describes the code generation assembler options for HCS08.

Table 4.10 Tool Settings — Assembler > Code Generation Options

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

HCS08 Assembler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target (Figure 4.44).

Figure 4.44 Tool Settings — HCS08 Assembler > Messages

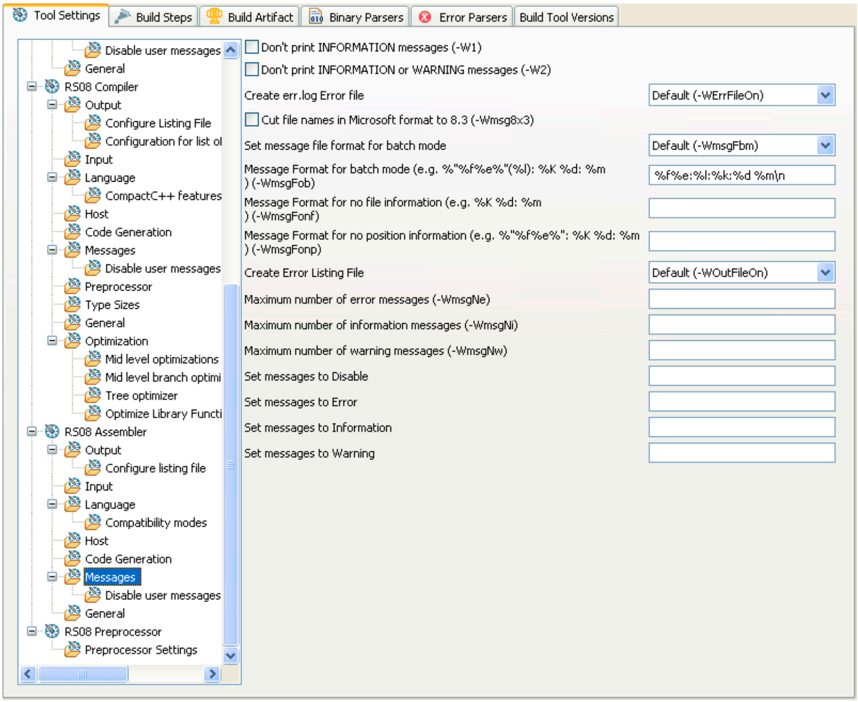


Table 4.11 lists and describes the message options.

Table 4.11 Tool Settings — HCS08 Assembler > Messages Options

Option	Description
Don't print `INFORMATION` messages (-`W1`)	Inhibits information message reporting. Only warning and error messages are generated.
Don't print `INFORMATION` or `WARNING` messages (`-W2`)	Suppresses all messages of type `INFORMATION` and `WARNING`. Only `ERROR` messages are generated.
Create `err.log` Error file	Using this option, the assembler uses a return code to report errors back to the tools. When errors occur, 16-bit window environments use err.log files, containing a list of error numbers, to report the errors. If no errors occur, the 16-bit window environments delete the err.log file.

Table 4.11 Tool Settings — HCS08 Assembler > Messages Options (*continued*)

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

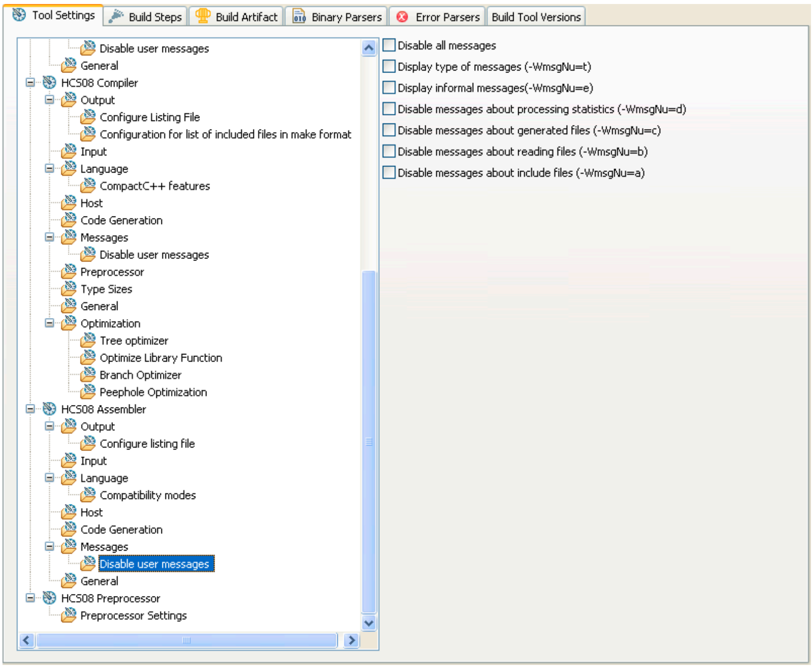
Table 4.11 Tool Settings — HCS08 Assembler > Messages Options (*continued*)

Option	Description
Set messages to Disable	Enter the messages that you want to disable.
Set messages to Error	Enter the messages that you want to set as error.
Set messages to Information	Enter the messages that you want to set as information.
Set messages to Warning	Enter the messages that you want to set as warning.

HCS08 Assembler > Messages > Disable user messages

Use this panel to specify the options for disabling the user messages for the HC(S)08 assembler. Figure 4.45 shows the **Disable user messages** panel.

Figure 4.45 Tool Settings — HCS08 Assembler > Messages > Disable user messages



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

HCS08 Assembler > General

Use this panel to specify the general assembler behavior.

Figure 4.46 shows the **General** panel.

Figure 4.46 Tool Settings — HCS08 Assembler > General

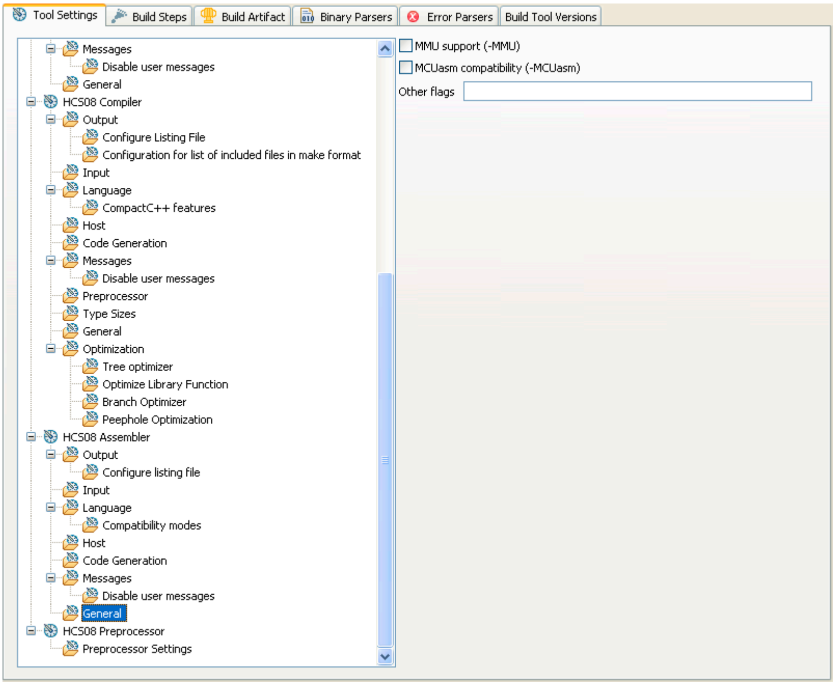


Table 4.12 lists and describes the general assembler options for HCS08.

Table 4.12 Tool Settings — Assembler > General Options

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

RS08 Assembler Build Properties Panels

Table 4.13 lists the build properties panels for the RS08 Assembler.

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

Table 4.13 Build Properties for RS08 Assembler

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

RS08 Assembler

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

Figure 4.47 shows the **Assembler** settings.

Figure 4.47 Tool Settings — RS08 Assembler

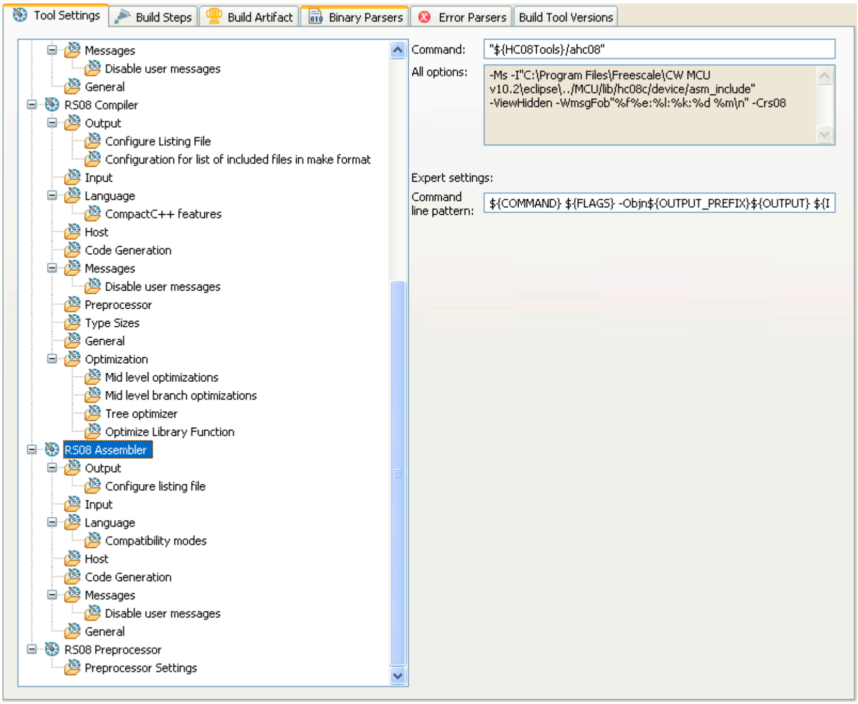


Table 4.14 lists and describes the assembler options for RS08.

Table 4.14 Tool Settings — Assembler Options

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

RS08 Assembler > Output

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

Figure 4.48 shows the **Output** panel.

Figure 4.48 Tool Settings — RS08 Assembler > Output

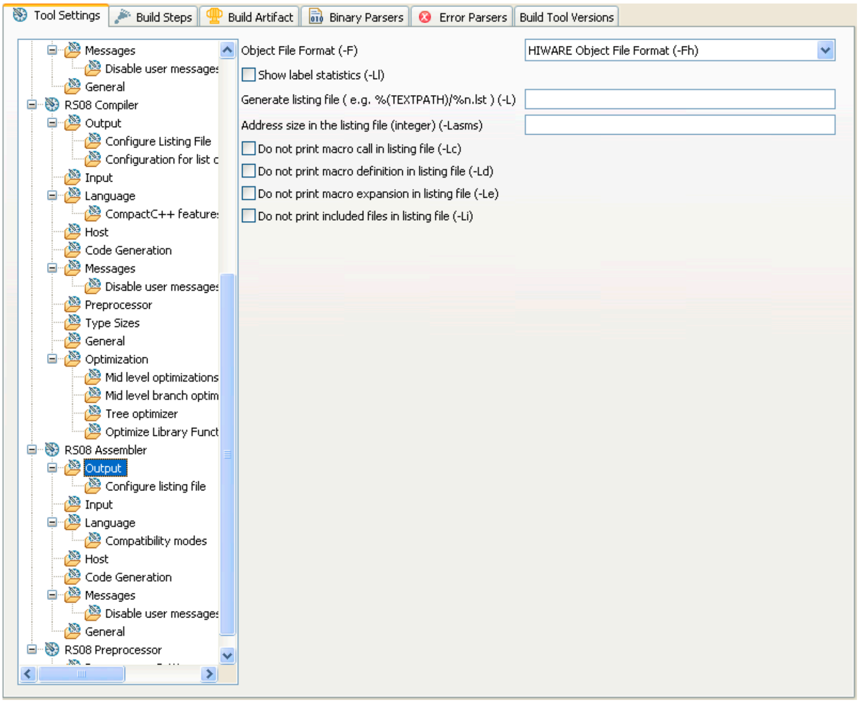


Table 4.15 lists and describes the output options for RS08 Assembler.

Table 4.15 Tool Settings — RS08 Assembler > Output Options

Option	Description
Object File Format (`-F`)	Defines the format for the output file generated by the Assembler.
Show label statistics (`-Li`)	Using the -Ll option, the assembler appends statistical information about the compilation session to the specified file. The information includes assembler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) for each procedure of the compiled file. The assembler appends the information to the specified filename (or the file make.txt, if no argument given). Set the TEXTPATH: Text File Path environment variable to store the file into the path specified by the environment variable. Otherwise the assembler stores the file in the current directory.

Table 4.15 Tool Settings — RS08 Assembler > Output Options (*continued*)

Option	Description
Generate listing file (e.g. %(TEXTPATH)/ %n.lst) (`-L`)	The -Lasm option causes the assembler to generate an assembler listing file directly. The assembler also prints all assembler-generated instructions to this file. The option specifies the name of the file. If no name is specified, the assembler takes a default of %n.lst. If the resulting filename contains no path information the assembler uses the TEXTPATH: Text File Path environment variable. The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, use this option only to review the generated code. It cannot currently be used to generate a file for assembly.
Address size in the listing file (`-Lasms`)	Specifies the size of the addresses displayed in the listing. Options are: • 1 to display addresses as xx • 2 to display addresses as xxxx • 3 to display addresses as xxxxxx • 4 to display addresses asf xxxxxxxx
Do not print macro call in listing file (`-Lc`)	Specifies whether macro calls encountered in the source code are expanded and appear in the listing file.
Do not print macro definition in listing file (`-Ld`)	Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.
Do not print macro expansion in listing file (`-Le`)	Switches on the generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definition and invocation lines as well as expanded include files.
Do not print included files in listing file (`-Li`)	Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

RS08 Assembler > Output > Configure Listing File

Use this panel to configure the listing file options of RS08 assembler. Figure 4.49 shows the **Configure Listing File** panel.

Figure 4.49 Tool Settings — RS08 Assembler > Output > Configure Listing File

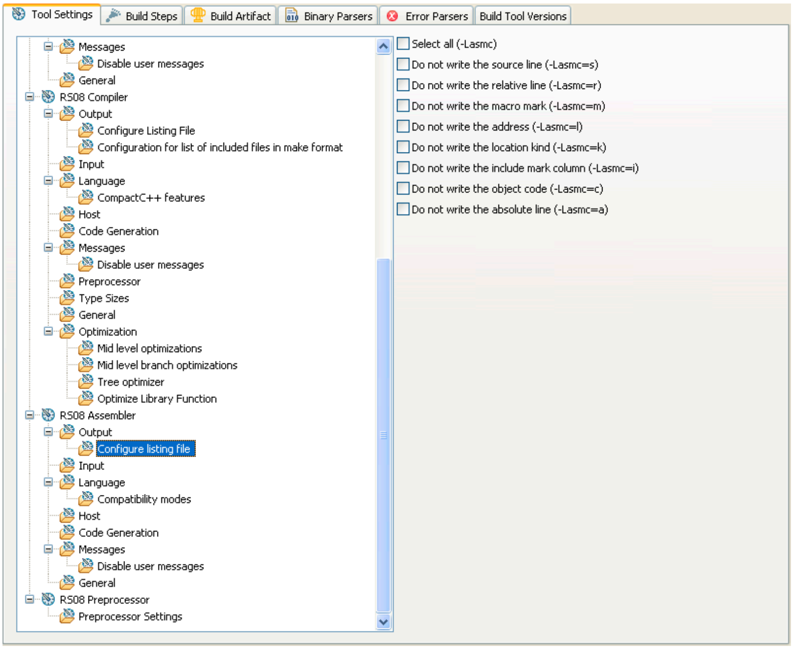


Table 4.16 lists and describes the Configure Listing File options for RS08 Assembler.

Table 4.16 Tool Settings — RS08 Assembler > Configure Listing File Options

Option	Description
Select all (`-Lasmc`)	Print all the columns in the listing file.
Do not write the source line (`-Lasmc=s`)	Do not print source column in the listing file.
Do not write the relative line (`-Lasmc=r`)	Do not print relative column (Rel.) in the listing file.
Do not write the macro mark (`-Lasmc=m`)	Do not print macro mark column in the listing file.
Do not write the address (`-Lasmc=l`)	Do not print address column (Loc) in the listing file.
Do not write the location kind (`-Lasmc=k`)	Do not print the location type column in the listing file.

Table 4.16 Tool Settings — RS08 Assembler > Configure Listing File Options (*continued*)

Option	Description
Do not write the include mark column (`-Lasmc=i`)	Do not print the include mark column in the listing file.
Do not write the object code (`-Lasmc=c`)	Do not print the object code in the listing file.
Do not write the absolute line (`-Lasmc=a`)	Do not print the absolute column (Abs.) in the listing file.

RS08 Assembler > Input

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

Figure 4.50 shows the **Input** panel.

Figure 4.50 Tool Settings — RS08 Assembler > Input

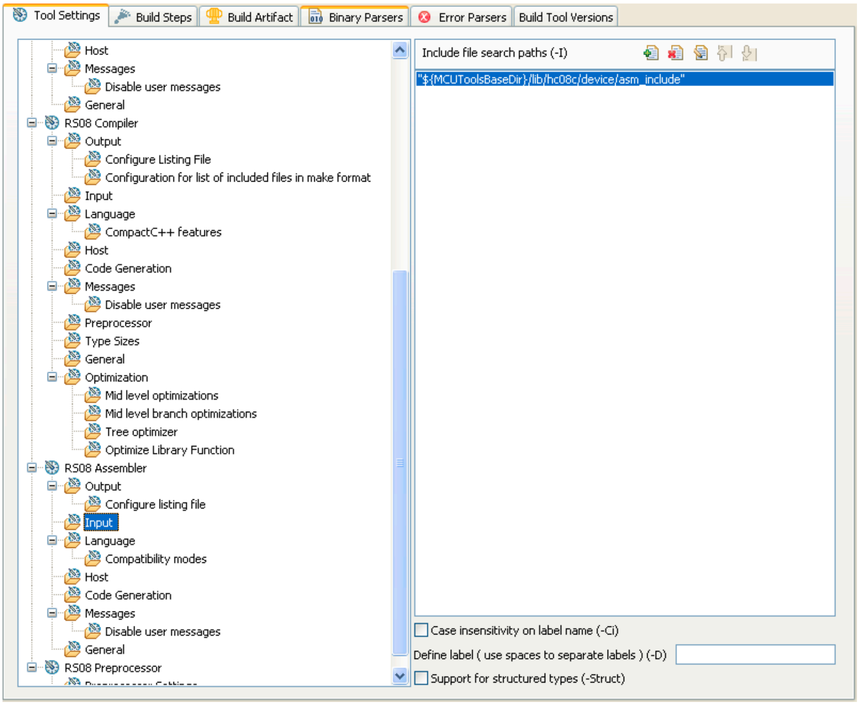


Table 4.17 lists and describes the input options of RS08 assembler.

Table 4.17 Tool Settings — Assembler > Input options

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

Table 4.18 lists and describes the toolbar buttons that help work with the file search paths.

Table 4.18 Search Paths Toolbar Buttons

Button	Description
	**Add —** Click to open the **Add directory path** dialog box and specify the file search path.
	**Delete —** Click to delete the selected file search path.
	**Edit —** Click to open the **Edit directory path** dialog box and update the selected object file search path.
	**Move up —** Click to move the selected file search path one position higher in the list.
	**Move down —** Click to move the selected file search path one position lower in the list.

RS08 Assembler > Language

Use this panel to specify code- and symbol-generation options for the RS08 assembler.

Figure 4.51 shows the **Language** panel.

Figure 4.51 Tool Settings — RS08 Assembler > Language

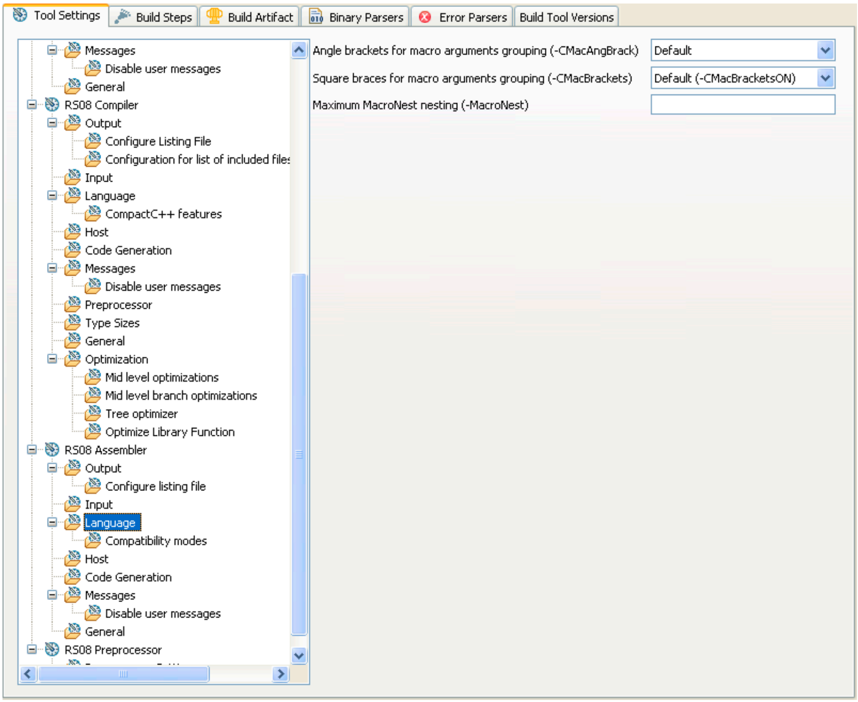


Table 4.19 lists and describes the language options for RS08 Assembler.

Table 4.19 Tool Settings — RS08 Assembler > Language Options

Option	Description
Angle brackets for macro arguments grouping (`-CMacAngBrack`)	Controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the [? ?] syntax instead. Options are: • Allow • Disallow
Square braces for macro arguments grouping (`-CMacBrackets`)	Controls the availability of the [? ?] syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for [?] in the macro invocation context. Options are: • `-CMacBracketsON` • `-CMacBracketsOFF`
Maximum MacroNest nesting (`-MacroNest`)	Controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

RS08 Assembler > Language > Compatibility modes

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

Figure 4.52 shows the **Compatibility modes** panel.

Figure 4.52 Tool Settings — RS08 Assembler > Compatibility modes

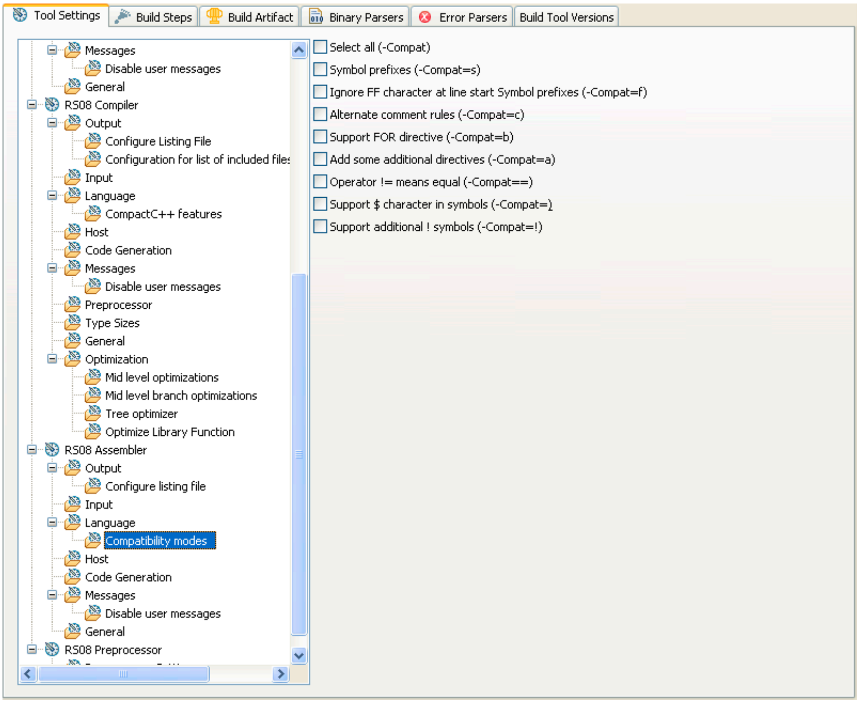


Table 4.20 lists and describes the compatibility mode options for RS08 Assembler.

Table 4.20 Tool Settings — RS08 Assembler > Compatibility modes Options

Option	Description
Select all (`-Compat`)	Check to enable all compatibility mode options.
Symbol prefixes (`-Compat=s`)	With this suboption, the Assembler accepts "pgz:" and "byte:" prefixed for symbols in `XDEFs` and `XREFs`. They correspond to `XREF.B` or `XDEF.B` with the same symbols without the prefix.
Ignore FF character at line start Symbol prefixes (`-Compat=f`)	With this suboption, an otherwise improper character recognized from feed character is ignored.

Table 4.20 Tool Settings — RS08 Assembler > Compatibility modes Options (*continued*)

Option	Description
Alternate comment rules (`-Compat=c`)	With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler does issue a warning if such a comment does not start with a "*" or a ";".
Support FOR directive (`-Compat=b`)	With this suboption, the Assembler supports a FOR - Repeat assembly block assembly directive to generate repeated patterns more easily without having to use recursive macros.
Add some additional directives (`-Compat=a`)	With this suboption, some additional directives are added for enhanced compatibility. The Assembler actually supports a SECT directive as an alias of the usual SECTION - Declare Relocatable Section assembly directive. The SECT directive takes the section name as its first argument.
Operator != means equal (`-Compat==`)	The Assembler takes the default value of the != operator as not equal, as it is in the C language. For compatibility, this behavior can be changed to equal with this option. Because of the risks involved with this option for existing code, a message is issued for every != which is treated as equal.

Table 4.20 Tool Settings — RS08 Assembler > Compatibility modes Options (*continued*)

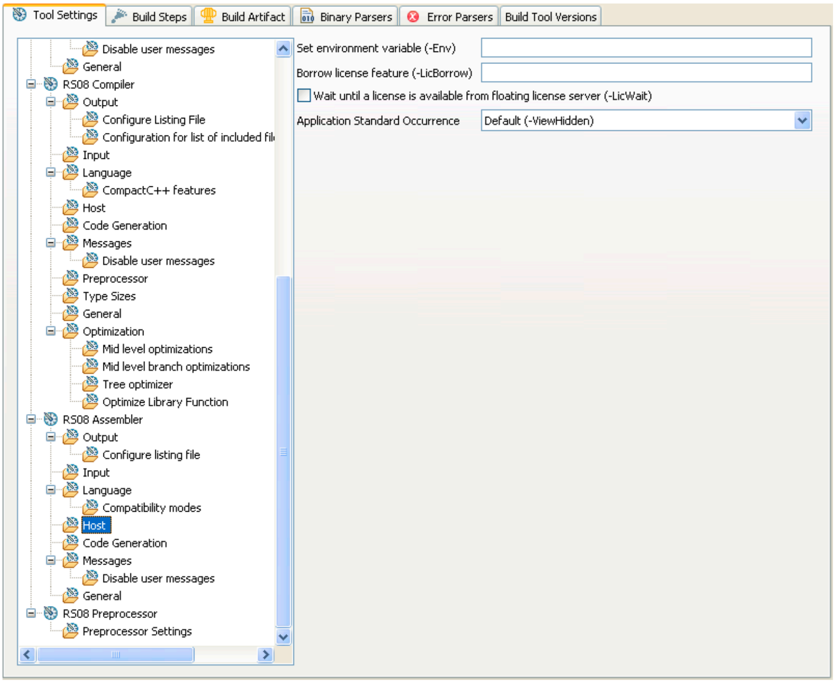
Option	Description
Support $ character in symbols (`-Compat=`)	With this suboption, the Assembler supports to start identifiers with a $ sign.
Support additional ! symbols (`-Compat=!`)	The following additional operators are defined when this option is used: • !^: exponentiation • !m: modulo • !@: signed 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

RS08 Assembler > Host

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

Figure 4.42 shows the **Host** settings.

Figure 4.53 Tool Settings — Host



[Table 4.9](#) lists and describes the memory model options for RS08.

Table 4.21 Tool Settings — RS08 Assembler > Host

Option	Description
Set environment variable (`-Env`)	This option sets an environment variable. Use this environment variable in the maker, or use to overwrite system environment variables.
Borrow license feature (`-LicBorrow`)	This option allows you to borrow a license feature until a given date or time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

Table 4.21 Tool Settings — RS08 Assembler > Host (*continued*)

Option	Description
Wait until a license is available from floating license server (`-LicWait`)	By default, if a license is not available from the floating license server, then the application will immediately return. With -LicWait set, the application will wait (blocking) until a license is available from the floating license server.
Application Standard Occurrence	This option allows you to select the standard appearance for the application window. By default the option **`-ViewHidden`** is selected. For more Information, refer to the section -View: Application standard occurrence.

RS08 Assembler > Code Generation

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

Figure 4.54 shows the **Code Generation** panel.

Figure 4.54 Tool Settings — RS08 Assembler > Code Generation

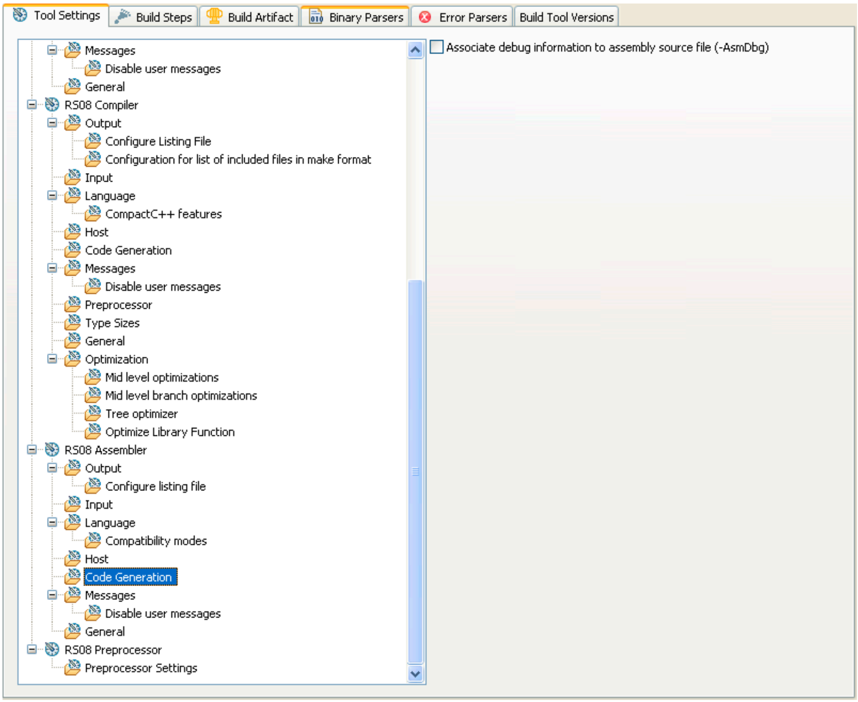


Table 4.22 lists and describes the Code Generation options for RS08 Assembler.

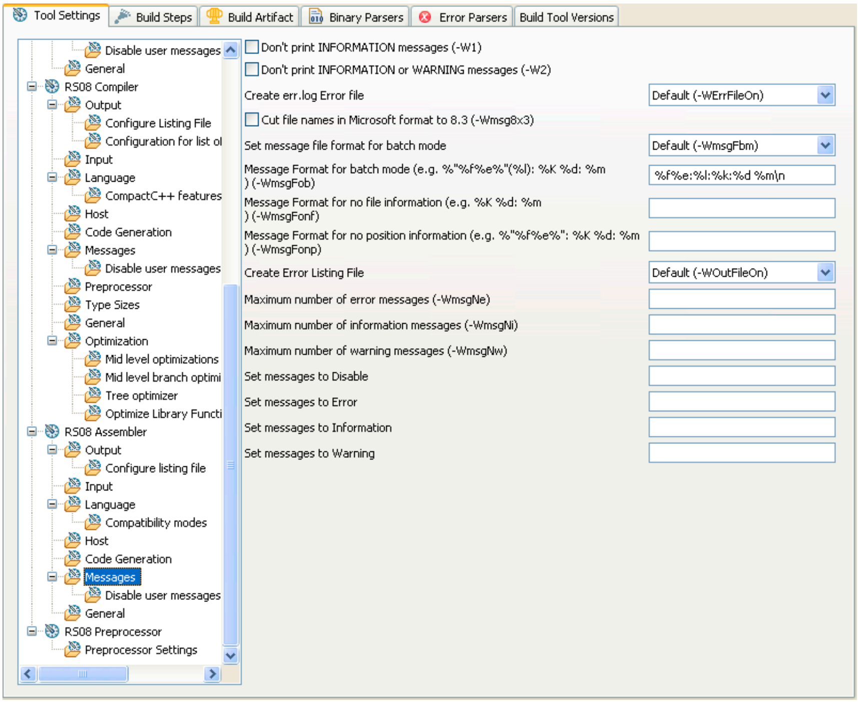
Table 4.22 Tool Settings — RS08 Assembler > Code Generation Options

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

RS08 Assembler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target (Figure 4.44).

Figure 4.55 Tool Settings — RS08 Assembler > Messages



[Table 4.11](#) lists and describes the message options.

Table 4.23 Tool Settings — RS08 Assembler > Messages Options

Option	Description
Don't print `INFORMATION` messages (`-W1`)	Inhibits information message reporting. Only warning and error messages are generated.
Don't print `INFORMATION` or `WARNING` messages (`-W2`)	Suppresses all messages of type `INFORMATION` and `WARNING`. Only `ERROR` messages are generated.
Create `err.log` Error file	Using this option, the assembler uses a return code to report errors back to the tools. When errors occur, 16-bit window environments use err.log files, containing a list of error numbers, to report the errors. If no errors occur, the 16-bit window environments delete the err.log file.

Table 4.23 Tool Settings — RS08 Assembler > Messages Options (*continued*)

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

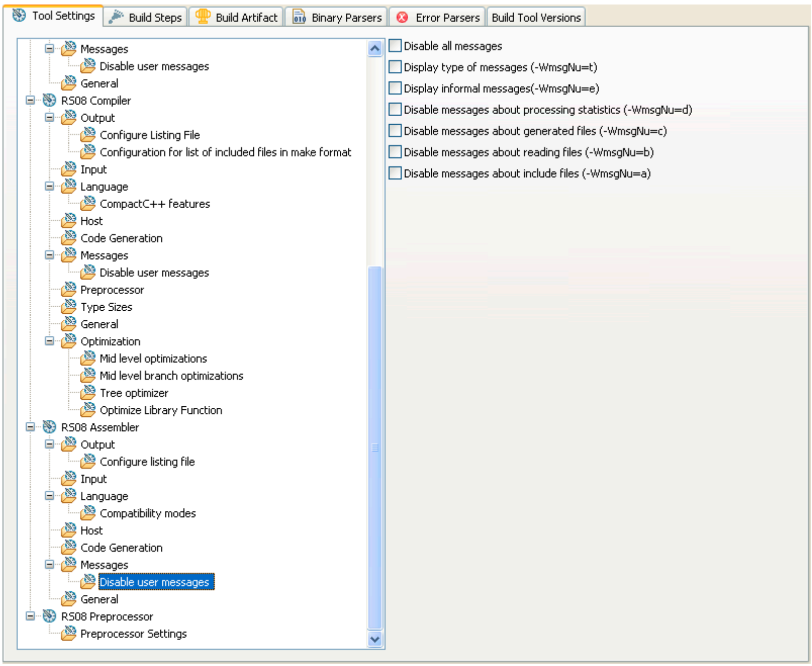
Table 4.23 Tool Settings — RS08 Assembler > Messages Options (*continued*)

Option	Description
Set messages to Disable	Enter the messages that you want to disable.
Set messages to Error	Enter the messages that you want to set as error.
Set messages to Information	Enter the messages that you want to set as information.
Set messages to Warning	Enter the messages that you want to set as warning.

RS08 Assembler > Messages > Disable user messages

Use this panel to specify the options for disabling the user messages for the RS08 assembler. Figure 4.45 shows the **Disable user messages** panel.

Figure 4.56 Tool Settings — RS08 Assembler > Messages > Disable user messages



NOTE For information about the options available in the **Disable user messages** panel of RS08 assembler, refer to the -WmsgNu: Disable user messages.

RS08 Assembler > General

Use this panel to specify the general assembler behavior.

Figure 4.57 shows the **General** panel.

Figure 4.57 Tool Settings — RS08 Assembler > General

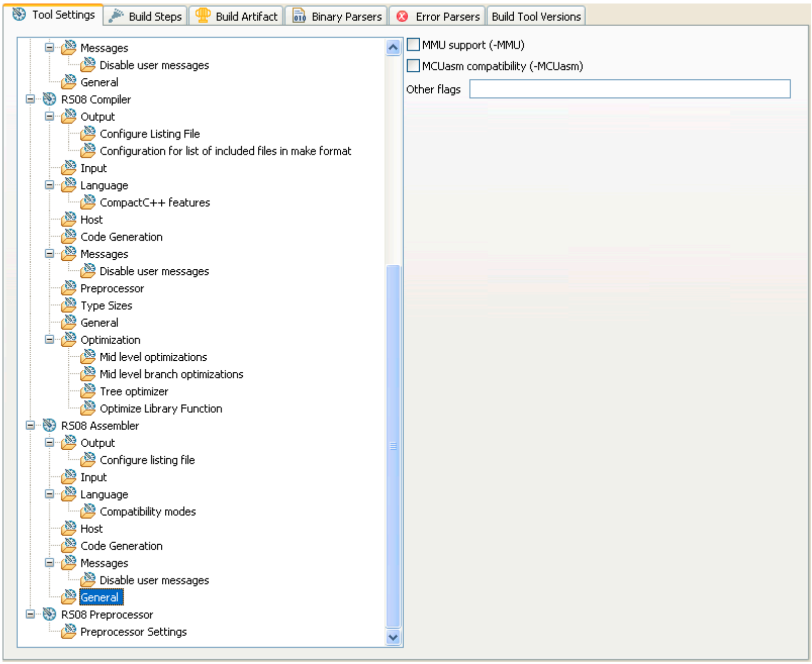


Table 4.24 lists and describes the general assembler options for RS08.

Table 4.24 Tool Settings — Assembler > General Options

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

5

Assembler Graphical User Interface

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

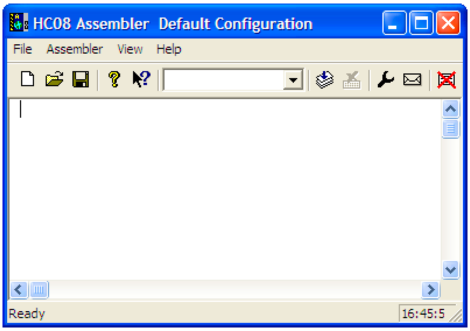
This chapter covers the following topics:

- Starting Assembler
- Assembler Main Window
- Editor Setting Dialog Box
- Save Configuration Dialog Box
- Option Settings dialog box
- Message Settings Dialog Box
- About Dialog Box
- Specifying Input File
- Message/Error Feedback

Starting Assembler

When you start the Assembler, the Assembler displays a standard **Tip of the Day** dialog box containing news and tips about the Assembler (Figure 5.1).

Figure 5.1 Tip of the Day dialog box



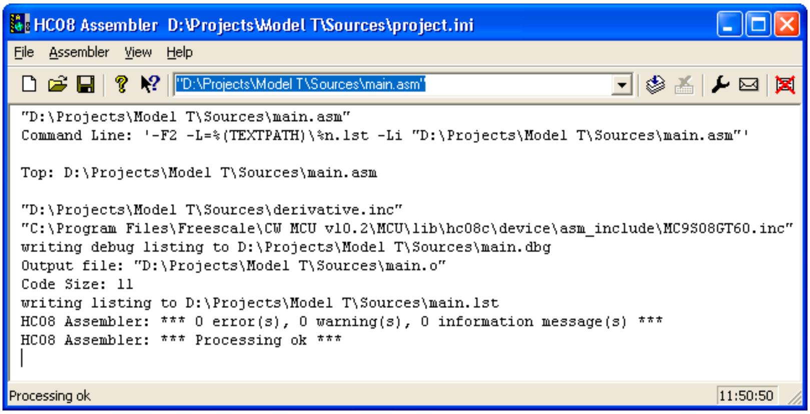
1. Click **Next Tip** to see the next piece of information about the Assembler.
2. Click **Close** to close the **Tip of the Day** dialog box.
   a. If you do not want the Assembler to automatically open the standard **Tip of the Day** dialog box when the Assembler is started, clear the **Show Tips on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

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 5.2).

Figure 5.2 Microcontroller 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.

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

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

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

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

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

Figure 5.3 Status Bar

Assembler Menu Bar

The following menus are available in the menu bar (Table 5.1):

Table 5.1 Menu bar options

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

File Menu

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

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

Table 5.2 File Menu Options

Menu Entry	Description
Assemble	A standard **Open File** dialog box is opened, displaying the list of all the `*.asm` files in the project directory. The input file can be selected using the features from the standard **Open File** dialog box. The selected file is assembled when the **Open File** dialog box is closed by clicking **OK**.
New/Default Configuration	Resets the Assembler option settings to their default values. The default Assembler options which are activated are specified in the Assembler Options chapter.
Load Configuration	A standard **Open File** dialog box is opened, displaying the list of all the `*.ini` files in the project directory. The configuration file can be selected using the features from the standard **Open File** dialog box. The configuration data stored in the selected file is loaded and used in further assembly sessions.
Save Configuration	Saves the current settings in the configuration file specified on the title bar.

Table 5.2 File Menu Options (*continued*)

Menu Entry	Description
Save Configuration As...	A standard **Save As** dialog box is opened, displaying the list of all the `*.ini` files in the project directory. The name or location of the configuration file can be specified using the features from the standard **Save As** dialog box. The current settings are saved in the specified configuration file when the **Save As** dialog box is closed by clicking **OK**.
Configuration...	Opens the **Configuration** dialog box to specify the editor used for error feedback and which parts to save with a configuration. See Editor Setting Dialog Box and Save Configuration Dialog Box.
1. .... project.ini 2.	Recent project list. This list can be used to reopen a recently opened project.
Exit	Closes the Assembler.

Assembler Menu

The **Assembler** menu (Table 5.3) allows you to customize the Assembler. You can graphically set or reset the Assembler options or to stop the assembling process.

Table 5.3 Assembler Menu Options

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

View Menu

The **View** menu (Table 5.4) 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 5.4 View Menu Options

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

Editor Setting Dialog Box

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

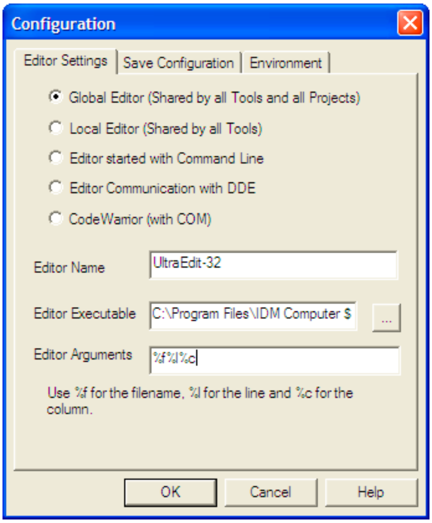
These are the main entries for the Editor configuration:

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

Global Editor (shared by all tools and projects)

This entry (Figure 5.4) is shared by all tools for all projects. This setting is stored in the `[Editor]` section of the `mcutools.ini` global initialization file. Some Modifiers can be specified in the editor command line.

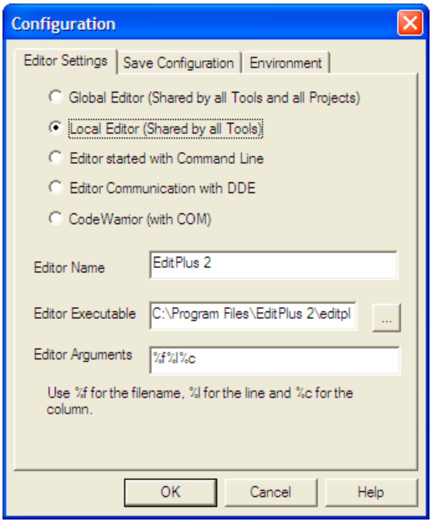
Figure 5.4 Global Editor Configuration Dialog Box



Local Editor (shared by all tools)

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

Figure 5.5 Local Editor Configuration Dialog Box



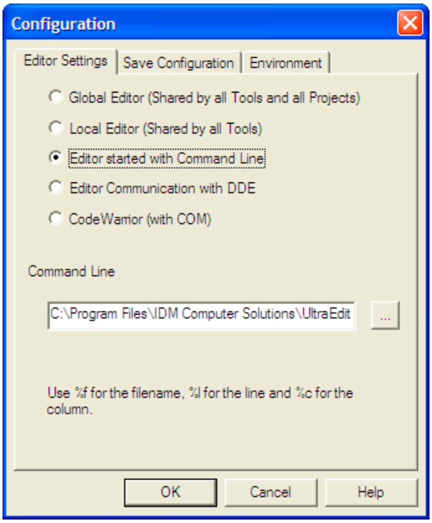
Editor Started with Command Line

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

Enter the command which should be used to start the editor (Figure 5.6).

The format from the editor command depends on the syntax which should be used to start the editor. Modifiers can be specified in the editor command line to refer to a filename and line and column position numbers. (See the Modifiers section below.)

Figure 5.6 Command Line Editor Configuration



Example of Configuring a Command Line Editor

The following case portrays the syntax used for configuring an external editors. Listing 5.1 can be used for the UltraEdit-32 editor.

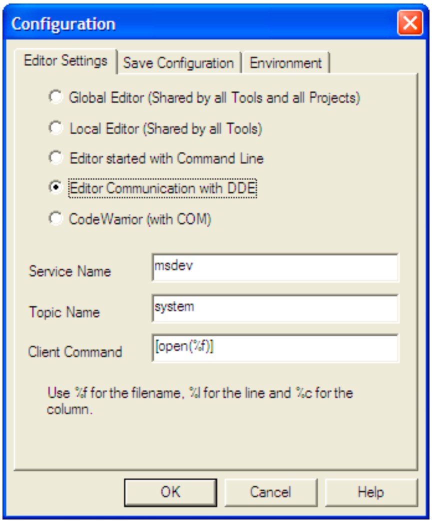
Listing 5.1 UltraEdit-32 configuration

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

Editor Started with DDE

Enter the service, topic and client name to be used for a Dynamic Data Exchange (DDE) connection to the editor (Figure 5.7). All entries can have modifiers for the filename and line number, as explained in the Modifiers section.

Figure 5.7 DDE Editor Configuration



For the Microsoft Developer Studio, use the settings in Listing 5.2:

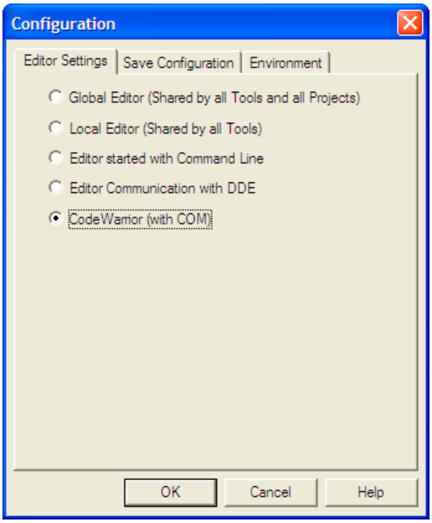
Listing 5.2 Microsoft Developer Studio configuration settings

```
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 5.8).

Figure 5.8 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.

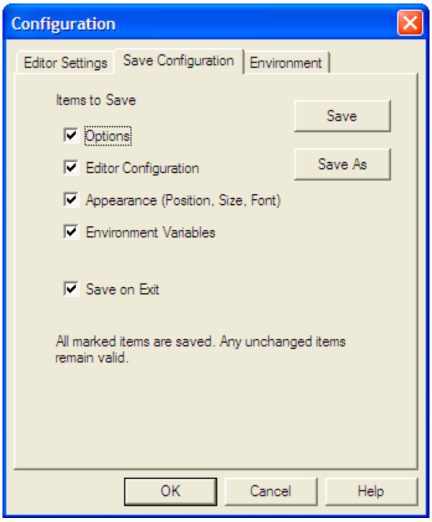
CAUTION The `%l` modifier can only be used with an editor which can be started with a line number as a parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When you work with such an editor, you can start it with the filename as a parameter and then select the menu entry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

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

Save Configuration Dialog Box

The second index of the configuration dialog box contains all options for the save operation (Figure 5.9).

Figure 5.9 Save Configuration Dialog Box



In the **Save Configuration** index, there are four checkboxes 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.

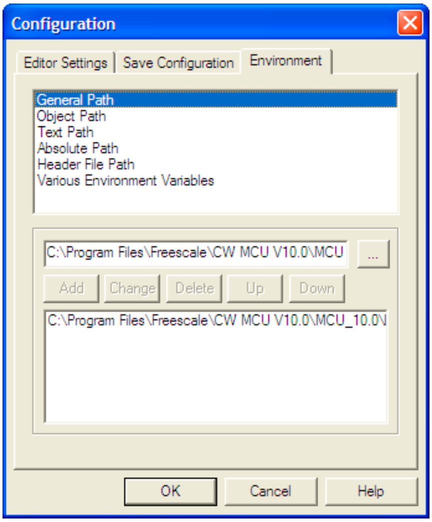
NOTE Almost all settings are stored in the project configuration file. The only exceptions are:
- The recently used configuration list.
- All settings in the Save Configuration dialog box.

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 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 box is used to configure the environment (Figure 5.10).

Figure 5.10 Environment Configuration Dialog Box



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

The following variables are available (Table 5.5):

Table 5.5 Path Environment Variables

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

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

The following buttons are available for the Configuration dialog box:

- **Add**: Adds a new line or entry

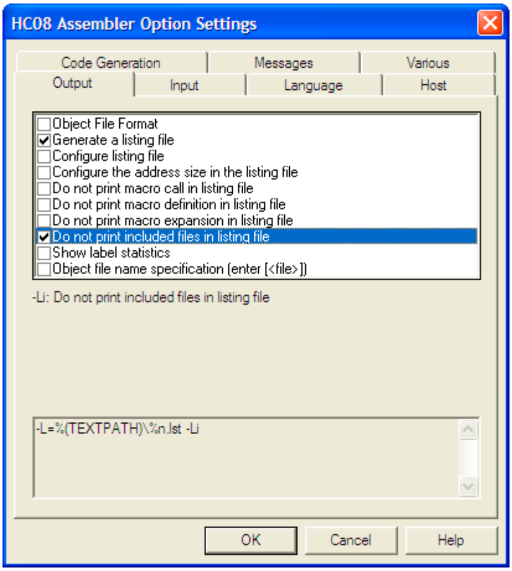
- **Change**: Changes a line or entry
- **Delete**: Deletes a line or entry
- **Up**: Moves a line or entry up
- **Down**: Moves a line or entry down

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

Option Settings dialog box

Use this dialog box (Figure 5.11) to set or reset assembler options.

Figure 5.11 Option Settings dialog box



The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet (Table 5.6):

Table 5.6 Option Settings Options

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

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

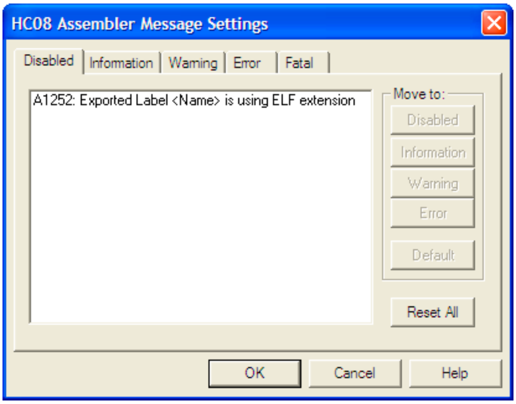
When the dialog box is opened and no option is selected, pressing the **F1** key or the **Help** button shows the help about this dialog box.

The available options are listed in the Assembler Options chapter.

Message Settings Dialog Box

You can use the **Message Settings** (Table 5.7) dialog box to map messages to a different message class.

Figure 5.12 Message Settings Dialog Box



Some buttons in the dialog box may be disabled. For example, if an option cannot be moved to an information message, the **Move to: Information** button is disabled. Table 5.7 lists the options available in the **Message Settings** dialog box:

Table 5.7 Message Settings Options

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

A tab is available in the dialog box for each message group (Table 5.8):

Table 5.8 Message Group

Message Group	Description
Disabled	Lists all disabled messages. That means that messages displayed in the tab page will not be displayed by the Assembler.
Information	Lists all information messages. Information messages informs about action taken by the Assembler.
Warning	Lists all warning messages. When such a message is generated, translation of the input file continues and an object file will be generated.
Error	Lists all error messages. When such a message is generated, translation of the input file continues, but no object file will be generated.
Fatal	Lists all fatal error messages. When such a message is generated, translation of the input file stops immediately. Fatal messages cannot be changed. They are only listed to call context help.

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

Changing the Class Associated with a Message

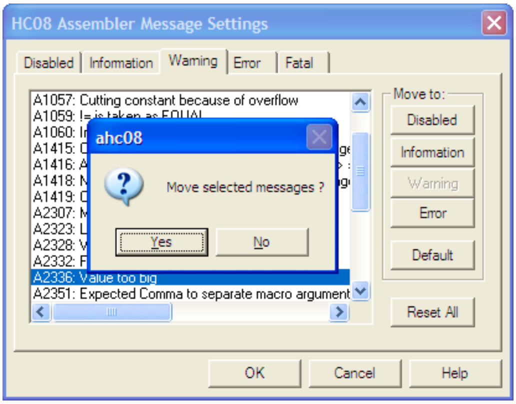
You can configure your own mapping of messages to the different classes. To do this, use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message, you have to select the message in the dialog box and then click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

Example

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

- Click the **Warning** tab to display the list of all warning messages.
- Click on the `A2336: Value too big` string in the to select the message.
- Click **Error** to define this message as an error message. The *Microcontroller* dialog box appears. Click **Yes** to close the dialog box (Figure 5.13).

Figure 5.13 Microcontroller Dialog Box



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

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

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

About Dialog Box

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

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

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

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

Specifying Input File

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

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

Use Command Line in Toolbar to Assemble

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

Assembling a New File

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

Assembling a File which has Already been Assembled

The commands executed previously can be displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file will be processed when the button **Assemble** in the toolbar is selected.

Use File > Assemble Entry

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

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 shown in Listing 5.3. A typical error message is like the one in Listing 5.4.

Listing 5.3 Typical error feedback message

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

Listing 5.4 Error message example

```
>> in "C:\Freescale\demo\fiboerr.asm", line 18, col 0, pos 722
         DC     label
               ^
ERROR A1104: Undeclared user defined symbol: label
```

For different message formats, see the following Assembler options:

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

Use Information from Assembler Window

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

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

Use User-defined Editor

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

Line Number can be Specified on the Command Line

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

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:

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

6

Environment

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

There are three ways to specify an environment:

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 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 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 the environment variables. The syntax is always the same (Listing 6.1).

Listing 6.1 Syntax for setting environment variables

```
Parameter: KeyName=ParamDef
```

Listing 6.2 is a typical example of setting an environment variable.

Listing 6.2 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 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., IDE, Make Utility, etc.).

For the UNIX operating system, the current directory for an executable is also the current directory from where the binary file has been started.

For MS Windows-based operating systems, the current directory definition is quite complex:

- If the tool is launched using the File Manager/Explorer, the current directory is the location of the launched executable tool.
- If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon in its properties.
- If the tool is launched by dragging a file on the icon of the executable tool on the desktop, the directory on the desktop is the current directory.
- If the tool is launched by another launching tool with its own current directory specification (e.g., an editor as IDE, a Make utility, etc.), the current directory is the one specified by the launching tool.
- When a local project file is loaded, the current directory is set to the directory which contains the local project file. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for an assembly source file does not change the current directory.

To overwrite this behavior, the DEFAULTDIR: Default current directory system environment variable may be used.

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

Environment macros

It is possible to use macros (Listing 6.3) in your environment settings.

Listing 6.3 Using a macro for setting environment variables

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

In the example in Listing 6.3, `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 6.4 are allowed. Note that they are case-sensitive and always surrounded by `{}`. Also the variable content contains a directory separator '\' as well.

`{Compiler}`

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

`{Project}`

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

`{System}`

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

Global initialization file - mctools.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 6.4 shows two typical locations used for the `mcutools.ini` files.

Listing 6.4 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.

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

The current directory is always the directory that holds the configuration file. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the whole `default.env` file is also reloaded. When a configuration file is loaded or stored, the options located in the ASMOPTIONS: Default assembler options environment variable are reloaded and added to the project's options.

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

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

Most environment variables contain path lists telling where to look for files. A path list is a list of directory names separated by semicolons following the syntax in Listing 6.5.

Listing 6.5 Syntax used for setting path lists of environment variables

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

Listing 6.6 is a typical example of setting an environment variable.

Listing 6.6 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 6.7 shows the use of an asterisk (`*`) for recursively searching the entire C drive for a configuration file with a `\INSTALL\LIB` path.

Listing 6.7 Recursive search for a continuation line

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

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

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

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

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

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 6.8):

Listing 6.8 Using multiple lines for an environment variable

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

Listing 6.8 is the same as the alternate source code in Listing 6.9.

Listing 6.9 Alternate form of using multiple lines

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

But this feature may be dangerous when used together with paths (Listing 6.10).

Listing 6.10 A path is included by the line continuation character

```
GENPATH=.\
TEXTFILE=.\txt
will result in
GENPATH=.TEXTFILE=.\txt
```

To avoid such problems, we recommend that you use a semicolon (`;`) at the end of a path if there is a backslash (`\`) at the end (Listing 6.11).

Listing 6.11 Recommended style whenever a backslash is present

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

Environment variables details

The remainder of this section is devoted to describing each of the environment variables available for the Assembler. The environment variables are listed in alphabetical order and each is divided into several sections (Table 6.1).

Table 6.1 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 chapter

COPYRIGHT: Copyright entry in object file

Tools

Compiler, Assembler, Linker, or Librarian

Syntax

```
COPYRIGHT=<copyright>
```

Arguments

`<copyright>`: copyright entry

Description

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

Example

```
COPYRIGHT=Copyright
```

See also

- USERNAME: User Name in object file
- INCLUDETIME: Creation time in the object file

DEFAULTDIR: Default current directory

Tools

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

Syntax

`DEFAULTDIR=<directory>`

Arguments

`<directory>`: Directory to be the default current directory

Description

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

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

Example

`DEFAULTDIR=C:\INSTALL\PROJECT`

See also

Current directory

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

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

Listing 6.12 Naming an error file

```
ERRORFILE=MyErrors.err
```

Listing 6.13 lists all errors into the `errors` file in the `\tmp` directory.

Listing 6.13 Naming an error file in a specific directory

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

Listing 6.14 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 6.14 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 6.15).

Listing 6.15 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 6.16).

Listing 6.16 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 6.17) shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called `EDOUT`:

Listing 6.17 Configuring error feedback with WinEdit

```
Installation directory: E:\INSTALL\prog
Project sources: D:\SRC
Common Sources for projects: E:\CLIB

Entry in default.env (D:\SRC\default.env):
ERRORFILE=E:\INSTALL\prog\EDOUT

Entry in WinEdit.ini (in Windows directory):
OUTPUT=E:\INSTALL\prog\EDOUT
```

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

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

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

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 type of S-Record File. If you specify `S1` while your code is loaded above `0xFFFF`, the S-Record File generated will not be correct because the addresses will all be truncated to 2-byte values.

When this variable is not set, the type of S-Record File generated will depend on the size of the address, which must be loaded there. If the address can be coded on 2 bytes, an `S1` record is generated. If the address is coded on 3 bytes, an `S2` record is generated. Otherwise, an `S3` record is generated.

Example

```
SRECORD=S2
```

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 section

USERNAME: User Name in object file

Tools

Compiler, Assembler, Linker, or Librarian

Syntax

`USERNAME=<user>`

Arguments

`<user>`: Name of user

Description

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

Example

`USERNAME=PowerUser`

See also

- COPYRIGHT: Copyright entry in object file
- INCLUDETIME: Creation time in the object file

7

Files

This chapter covers these topics:

- Input files
- Output files
- File processing

Input files

Input files to the Assembler:

- Source files
- Include files

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

Include files

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

Output files

Output files from the Assembler:

- Object files
- Absolute files
- S-Record Files
- Listing files
- Debug listing files
- Error listing file

Object files

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

Absolute files

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

S-Record Files

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

- If `SRECORD` = `S1`, the S-Record File gets the `*.s1` extension.
- If `SRECORD` = `S2`, the S-Record File gets the `*.s2` extension.

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

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

Listing files

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

Debug listing files

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

Error listing file

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

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

If the Assembler is started from the *IDE* (with '`%f`' given on the command line) or CodeWright (with '`%b%e`' given on the command line), this error file is not produced. Instead, it writes the error messages in a special Microsoft default format in a file called

EDOUT. Use *WinEdit*'s *Next Error* or CodeWright's *Find Next Error* command to see both error positions and the error messages.

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)

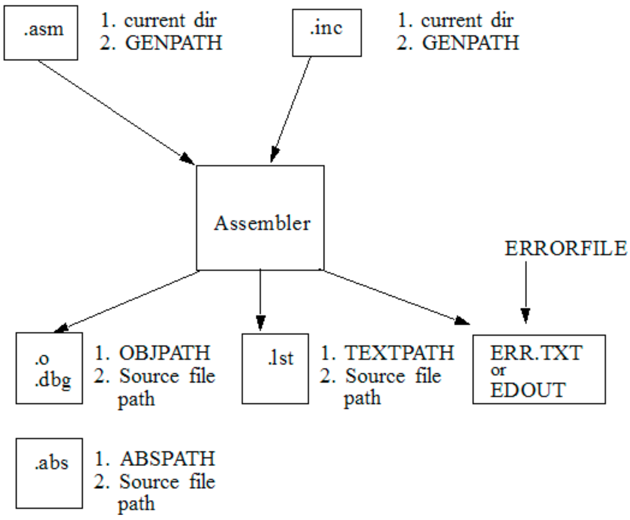
If ERRORFILE is set, the Assembler creates a message file named as specified in this environment variable.

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

File processing

Figure 7.1 shows the priority levels for the various files used by the Assembler.

Figure 7.1 Files used with the Assembler



8

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 hyphen (`-`) followed by one or more letters or digits. Anything not starting with a hyphen is supposed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the ASMOPTIONS: Default assembler options (Table 8.1) environment variable. Typically, each Assembler option is specified only once per assembling session.

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

NOTE It is not possible to combine options in different groups, e.g., `-Lc -W1` *cannot* be abbreviated by the terms `-LC1` or `-LCW1`.

Table 8.1 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 8.2) are grouped by:

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

Table 8.2 Assembler option categories

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

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

Each option has also a scope (Table 8.3).

Table 8.3 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 8.4).

Table 8.4 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 8.5)

Table 8.5 Special modifiers for assembler options

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

Examples using special modifiers

The assumed path and filename (filename base for the modifiers) used for the examples Listing 5.2 through Listing 5.13 is displayed in Listing 8.1.

Listing 8.1 Example filename and path used for the following examples

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

Using the `%p` modifier as in Listing 8.2 displays the path with a file separator but without the filename.

Listing 8.2 %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 8.3).

Listing 8.3 %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 8.4.

Listing 8.4 %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 8.5).

Listing 8.5 %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 8.6).

Listing 8.6 %e is used for returning the whole extension

```
myExt
```

The `%f` modifier returns the path and the filename without the file extension (Listing 8.7).

Listing 8.7 %f gives the path plus the filename (no file extension)

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

The path in Listing 8.1 contains a space, therefore using `%"` or `%'` is recommended (Listing 8.8 or Listing 8.9).

Listing 8.8 Use %”%f%” in case there is a space in its path, filename, or extension

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

Listing 8.9 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 8.10, then `$(TEXTPATH)\myfile.txt` is expressed as in Listing 8.11.

Listing 8.10 Example for setting TEXTPATH

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

Listing 8.11 $(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 8.12).

Listing 8.12 $(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 8.13.

Listing 8.13 %% allows a percent sign to be expressed

```
myExt%
```

List of Assembler option

The Table 8.6 lists each command line option you can use with the Assembler.

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

Table 8.6 Assembler options

Assembler option
-ArgFile: Specify a file from which additional command line options will be read
-AsmDbg: Emit assembly source file information in debug sections
-Ci: Switch case sensitivity on label names OFF
-CMacAngBrack: Angle brackets for grouping Macro Arguments
-CMacBrackets: Square brackets for macro arguments grouping
-Compat: Compatibility modes
-CS08/-C08/-CRS08: Derivative family
-D: Define Label
-DefLabel: Improves support for data allocation directives
-Env: Set environment variable
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format
-H: Short Help
-I: Include file path
-L: Generate a listing file
-Lasmc: Configure listing file
-Lasms: Configure the address size in the listing file
-Lc: No Macro call in listing file
-Ld: No macro definition in listing file
-Le: No Macro expansion in listing file

Table 8.6 Assembler options (*continued*)

Assembler option
-Li: No included file in listing file
-Lic: License information
-LicA: License information about every feature in directory
-LicBorrow: Borrow license feature
-LicWait: Wait until floating license is available from floating License Server
-Ll: Show label statistics
-M (-Ms, -Mt): Memory model
-MacroNest: Configure maximum macro nesting
-MCUasm: Switch compatibility with MCUasm ON
-MMU: Enable Memory Management Unit (MMU) Support
-N: Display notify box
-NoBeep: No beep in case of an error
-NoDebugInfo: No debug information for ELF/DWARF files
-NoEnv: Do not use environment
-ObjN: Object filename specification
-Prod: Specify project file at startup
-Struct: Support for structured types
-V: Prints the Assembler version
-View: Application standard occurrence
-W1: No information messages
-W2: No information and warning messages
-WErrFile: Create "err.log" error file
-Wmsg8x3: Cut filenames in Microsoft format to 8.3
-WmsgCE: RGB color for error messages
-WmsgCF: RGB color for fatal messages

Table 8.6 Assembler options (*continued*)

Assembler option
-WmsgCI: RGB color for information messages
-WmsgCU: RGB color for user messages
-WmsgCW: RGB color for warning messages
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
-WmsgFob: Message format for batch mode
-WmsgFoi: Message format for interactive mode
-WmsgFonf: Message format for no file information
-WmsgFonp: Message format for no position information
-WmsgNe: Number of error messages
-WmsgNi: Number of Information messages
-WmsgNu: Disable user messages
-WmsgNw: Number of Warning messages
-WmsgSd: Setting a message to disable
-WmsgSe: Setting a message to Error
-WmsgSi: Setting a message to Information
-WmsgSw: Setting a Message to Warning
-WOutFile: Create error listing file
-WStdout: Write to standard output

Detailed listing of all assembler options

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

-ArgFile: Specify a file from which additional command line options will be read

Group

HOST

Scope

Function

Syntax

```
-ArgFile<filename>
```

Arguments

<filename>: Specify filename that has options to be passed to command line

Description

The options present in file are appended to existing command line options.

Example

```
option.txt
-M
Linker.exe -ArgFileoption.txt test.prm
```

This is equivalent to linker.exe -M test.prm and linker generates output file test.map

-AsmDbg: Emit assembly source file information in debug sec-

tions

Group

CODE GENERATION

Scope

Function

Syntax

`-AsmDbg`

Arguments

None

Description

This option when enabled, passes the assembly source file name information to DWARF sections. When the output .abs file is debugged, the actual assembly source file is displayed instead of intermediary <filename>.dbg file.

-Ci: Switch case sensitivity on label names OFF

Group

Input

Scope

Assembly Unit

Syntax

`-Ci`

Arguments

None

Default

None

Description

This option turns off case sensitivity on label names. When this option is activated, the Assembler ignores case sensitivity for label names. If the Assembler generates object files but not absolute files directly (`-FA2` assembler option), the case of exported or imported labels must still match. Or, the `-Ci` assembler option should be specified in the linker as well.

Example

When case sensitivity on label names is switched off, the Assembler will not generate an error message for the assembly source code in Listing 8.14.

Listing 8.14 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 assembler option

-CMacAngBrack: Angle brackets for grouping Macro Arguments

Group

Language

Scope

Application

Syntax

`-CMacAngBrack(ON|OFF)`

Arguments

`ON` or `OFF`

Default

None

Description

This option controls whether the < > syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for < in the macro invocation context. There are cases where the angle brackets are ambiguous. In new code, use the `[? ?]` syntax instead.

See also

Macro argument grouping

-CMacBrackets: Square brackets for macro arguments grouping option

-CMacBrackets: Square brackets for macro arguments grouping

Group

Language

Scope

Application

Syntax

`-CMacBrackets(ON|OFF)`

Arguments

`ON` or `OFF`

Default

`ON`

Description

This option controls the availability of the `[? ?]` syntax for macro invocation argument grouping. When it is disabled, the Assembler does not recognize the special meaning for `[?` in the macro invocation context.

See also

Macro argument grouping

-CMacAngBrack: Angle brackets for grouping Macro Arguments option

-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 does issue a warning if such a comment does not start with a `"*"` or a `";"`.

Examples

Listing 8.15 demonstrates that when `-Compat=c`, comments can start with a `*`.

Listing 8.15 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 8.16 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 8.16 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 feed character is ignored.

- `$`: Support the $ character in symbols

  With this suboption, the Assembler supports to start identifiers with a `$` sign.

- `a`: Add some additional directives

  With this suboption, some additional directives are added for enhanced compatibility.

  The Assembler actually supports a `SECT` directive as an alias of the usual SECTION - Declare Relocatable Section assembly directive. The `SECT` directive takes the section name as its first argument.

- `b`: support the `FOR` directive

  With this suboption, the Assembler supports a FOR - Repeat assembly block assembly directive to generate repeated patterns more easily without having to use recursive macros.

-CS08/-C08/-CRS08: Derivative family

Group

Code Generation

Scope

Application

Syntax

`-C08|-CS08|-CRS08`

Arguments

None

Default

`-C08`

Description

The Assembler supports three different HC08-derived cores. The HC08 itself (`-C08`), the enhanced HCS08 (`-CS08`), and the RS08 (`-CRS08`).

The HCS08 family supports additional addressing modes for the `CPHX`, `LDHX`, and `STHX` instructions and also a new BGND instruction. All these enhancements are allowed when the `-CS08` option is specified. All instructions and addressing modes available for the HC08 are also available for the HCS08 so that this core remains binary compatible with its predecessor.

The RS08 family does not support all instructions and addressing modes of the HC08. Also, the encoding of the supported instructions is not binary compatible.

Table 8.7 Table of new instructions or addressing modes for the HCS08

Instruction	Addr. mode	Description
LDHX	EXT	load from a 16-bit absolute address
	IX	load HX via 0,X
	IX1	load HX via 1,X...255,X
	IX2	load HX via old HX+ any offset
	SP1	load HX from stack
STHX	EXT	store HX to a 16-bit absolute address
	SP1	store HX to stack
CPHX	EXT	compare HX with a 16-bit address
	SP1	compare HX with the stack
BGND		enter the Background Debug Mode

-D: Define Label

Group

Input

Scope

Assembly Unit

Syntax

`-D<LabelName>[=<Value>]`

Arguments

`<LabelName>`: Name of label.

`<Value>`: Value for label. `0` if not present.

Default

`0` for `Value`.

Description

This option behaves as if a `Label: EQU Value` is at the start of the main source file. When no explicit value is given, 0 is used as the default.

This option can be used to build different versions with one common source file.

Example

Conditional inclusion of a copyright notice. See Listing 8.17 and Listing 8.18.

Listing 8.17 Source code that conditionally includes a copyright notice

```
YearAsString: MACRO
   DC.B $30+(\1 /1000)%10
   DC.B $30+(\1 / 100)%10
   DC.B $30+(\1 /  10)%10
   DC.B $30+(\1 /   1)%10
ENDM

  ifdef ADD_COPYRIGHT
   ORG $1000
   DC.B "Copyright by "
   DC.B "John Doe"
  ifdef YEAR
   DC.B " 1999-"
   YearAsString YEAR
  endif
   DC.B 0
  endif
```

When assembled with the option `-dADD_COPYRIGHT -dYEAR=2005`, Listing 8.18 is generated:

Listing 8.18 Generated list file

```
 1    1                        YearAsString:  MACRO
 2    2                            DC.B $30+(\1 /1000)%10
 3    3                            DC.B $30+(\1 / 100)%10
 4    4                            DC.B $30+(\1 /  10)%10
 5    5                            DC.B $30+(\1 /   1)%10
 6    6                        ENDM
 7    7
 8    8           0000 0001    ifdef ADD_COPYRIGHT
 9    9                          ORG $1000
10   10   a001000 436F 7079      DC.B "Copyright by "
           001004 7269 6768
           001008 7420 6279
           00100C 20
11   11   a00100D 4A6F 686E      DC.B "John Doe"
           001011 2044 6F65
12   12           0000 0001      ifdef YEAR
13   13   a001015 2031 3939        DC.B " 1999-"
           001019 392D
14   14                        YearAsString YEAR
15    2m  a00101B 32           +   DC.B $30+(YEAR /1000)%10
16    3m  a00101C 30           +   DC.B $30+(YEAR / 100)%10
17    4m  a00101D 30           +   DC.B $30+(YEAR /  10)%10
18    5m  a00101E 31           +   DC.B $30+(YEAR /   1)%10
19   15                        endif
20   16   a00101F 00               DC.B 0
21   17                        endif
```

-DefLabel: Improves support for data allocation directives

Group

Input

Scope

Assembly Unit

Syntax

`-DefLabel`

Arguments

None

Default

None

Description

Improves support for data allocation directives. On passing this option, the data directives (not associated to any label) get associated with previous defined labels (if exists). This inhibits the emission of temporary variables (`VARx`) by assembler.

Example

Listing 8.19 shows the example for the `-DefLabel`.

Listing 8.19 Example -DefLabel

```
MySection: SECTION
TTab_TIT_45:    ; Modo 5
       DC.B    $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FA,$90,$20,$00,$01,$4F,$FF
,$FB,$FF,$FF,$FF,$FF,$FC
       DC.B    $C1,$FF,$FF,$FF,$FF,$FF,$FF,$F7,$64,$88,$00,$14,$BF,$FF
,$F5,$FF,$EF,$B8,$3F,$FE
       DC.B 5
       DC.W 3
mainLoop:
            RTS

With -DefLabel option OFF:
Output:
9-VAR00001         0   14 LOCAL FUNC      9 (MySection)
10-VAR00002       14   14 LOCAL FUNC      9 (MySection)
11-VAR00003       28   1 LOCAL  FUNC      9 (MySection)
12-VAR00004       29   3 LOCAL  FUNC      9 (MySection)

With -DefLabel option ON:
Output:
9-TTab_TIT_45      0   29 LOCAL FUNC      9 (MySection)
10-VAR00001        29  3  LOCAL FUNC      9 (MySection)
```

The input file when assembled with option `-DefLabel`, allocates the `DC.B` directives to symbol `TTab_TIT_45` and `DC.W` to dummy variable `VAR00001`.

-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

Environment variables details

-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format

Group

Output

Scope

Application

Syntax

`-F(h|2o|A2o|2|A2)`

Arguments

`h`: HIWARE object-file format; this is the default

`2o`: Compatible ELF/DWARF 2.0 object-file format

`A2o`: Compatible ELF/DWARF 2.0 absolute-file format

`2`: ELF/DWARF 2.0 object-file format

`A2`: ELF/DWARF 2.0 absolute-file format

Default

`-F2`

Description

Defines the format for the output file generated by the Assembler:

- With the `-Fh` option set, the Assembler uses a proprietary (HIWARE) object-file format.
- With the `-F2` option set, the Assembler produces an ELF/DWARF object file. This object-file format may also be supported by other Compiler or Assembler vendors.
- With the `-FA2` option set, the Assembler produces an ELF/DWARF absolute file. This file format may also be supported by other Compiler or Assembler vendors.

Note that the ELF/DWARF 2.0 file format has been updated in the current version of the Assembler. If you are using HI-WAVE version 5.2 (or an earlier version), `-F2o` or `-FA2o` must be used to generate the ELF/DWARF 2.0 object files which can be loaded in the debugger.

Example

```
ASMOPTIONS=-F2
```

NOTE For the RS08 the HIWARE object file format is not available.

-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 8.20 is a portion of the list produced by the `-H` option:

Listing 8.20 Example Help listing

```
...
MESSAGE:
-N        Show notification box in case of errors
-NoBeep   No beep in case of an error
-W1       Do not print INFORMATION messages
-W2       Do not print INFORMATION or WARNING messages
-WErrFile Create "err.log" Error File
...
```

-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).

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 8.21), 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 8.21, together with the code from an include file (Listing 8.22) generates the output listing in Listing 8.23.

Listing 8.21 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 8.22 Example source code from an include file

```
cpChar: MACRO
        LDA \1
        STA \2
        ENDM
```

Listing 8.23 Assembly output listing

```
Abs. Rel.   Loc    Obj. code  Source line
---- ----   ------ ---------  -----------
   1    1                              XDEF  Start
   2    2                     MyData:  SECTION
   3    3   000000            char1:   DS.B  1
   4    4   000001            char2:   DS.B  1
   5    5                              INCLUDE "macro.inc"
   6   1i                     cpChar:  MACRO
   7   2i                              LDA   \1
   8   3i                              STA   \2
   9   4i                              ENDM
  10    6                     CodeSec: SECTION
  11    7                     Start:
  12    8                              cpChar char1, char2
  13   2m   000000 C6 xxxx   +         LDA   char1
  14   3m   000003 C7 xxxx   +         STA   char2
  15    9   000006 9D                  NOP
```

The Assembler stores the content of included files in the listing file. The Assembler also stores macro definitions, invocations, and expansions in the listing file.

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

Assembler options:

- -Lasmc: Configure listing file
- -Lasms: Configure the address size in the listing file
- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file

-Lasmc: Configure listing file

Group

Output

Scope

Assembly unit

Syntax

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

Arguments

`s` - Do not write the source column

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

`m` - Do not write the macro mark

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

`k` - Do not write the location type

`i` - Do not write the include mark column

`c` - Do not write the object code

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

Default

Write all columns.

Description

The default-configured listing file shows a lot of information. With this option, the output can be reduced to columns which are of interest. This option configures which columns are printed in a listing file. To configure which lines to print, see the following assembler options: -Lc: No Macro call in listing file, -Ld: No macro definition in listing file, -Le: No Macro expansion in listing file, and -Li: No included file in listing file.

Example

For the following assembly source code, the Assembler generates the default-configured output listing (Listing 8.24):

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

```
DC.B 0
```

Listing 8.24 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 8.25:

Listing 8.25 Example output listing

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

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

Assembler options:

- -L: Generate a listing file
- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file
- -Lasms: Configure the address size in the listing file

-Lasms: Configure the address size in the listing file

Group

Output

Scope

Assembly unit

Syntax

`-Lasms{1|2|3|4}`

Arguments

1 - The address size is xx

2 - The address size is xxxx

3 - The address size is xxxxxx

4 - The address size is xxxxxxxx

Default

`-Lasms3`

Description

The default-configured listing file shows a lot of information. With this option, the size of the address column can be reduced to the size of interest. To configure which columns are printed, see the -Lasmc: Configure listing file option. To configure which lines to print, see the -Lc: No Macro call in listing file, -Ld: No macro definition in listing file, -Le: No Macro expansion in listing file, and -Li: No included file in listing file assembler options.

Example

For the following instruction:

```
NOP
```

the Assembler generates this default-configured output listing (Listing 8.26):

Listing 8.26 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 8.27 Example assembler output listing configured with -Lasms1

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

See also

Assembler Listing File chapter

Assembler options:

- -Lasmc: Configure listing file
- -L: Generate a listing file
- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file

-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 8.28, along with additional source code (Listing 8.29) from the `macro.inc` include file generates the output in the assembly listing file (Listing 8.30).

Listing 8.28 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 8.29 Example source code from the macro.inc file

```
cpChar:  MACRO
         LDA \1
         STA \2
         ENDM
```

Listing 8.30 Output assembly listing

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF Start
    2    2                      MyData:  SECTION
    3    3   000000             char1:   DS.B  1
    4    4   000001             char2:   DS.B  1
    5    5                      INCLUDE "macro.inc"
    6   1i                      cpChar:  MACRO
    7   2i                               LDA \1
    8   3i                               STA \2
    9   4i                               ENDM
   10    6                      CodeSec: SECTION
   11    7                      Start:
   13   2m   000000 C6 xxxx   +          LDA char1
   14   3m   000003 C7 xxxx   +          STA char2
   15    9   000006 9D                   NOP
```

The Assembler stores the content of included files in the listing file. The Assembler also stores macro definitions, invocations, and expansions in the listing file.

The listing file does not contain the line of source code that invoked the macro.

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

Assembler options:

- -L: Generate a listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file

-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 8.31 along with additional source code (Listing 8.32) from the `macro.inc` file generates an assembler output listing (Listing 8.33) file:

Listing 8.31 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 8.32 Example source code from an include file

```
cpChar:   MACRO
            LDA   \1
            STA   \2
          ENDM
```

Listing 8.33 Example assembler output listing

```
 Abs. Rel.    Loc    Obj. code  Source line
 ---- ----    ------ ---------  -----------
    1    1                               XDEF  Start
    2    2                      MyData:  SECTION
    3    3    000000            char1:   DS.B  1
    4    4    000001            char2:   DS.B  1
    5    5                                INCLUDE "macro.inc"
    6    1i                      cpChar:  MACRO
   10    6                       CodeSec: SECTION
   11    7                       Start:
   12    8                                cpChar char1, char2
   13    2m   000000 C6  xxxx   +         LDA char1
   14    3m   000003 C7  xxxx   +         STA char2
   15    9    000006 9D                   NOP
```

The Assembler stores that content of included files in the listing file. The Assembler also stores macro invocation and expansion in the listing file.

The listing file does not contain the source code from the macro definition.

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

Assembler options:

- -L: Generate a listing file
- -Lc: No Macro call in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file

-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 8.34 along with additional source code (Listing 8.35) from the `macro.inc` file generates an assembly output listing file (Listing 8.36):

Listing 8.34 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 8.35 Example source code from an included file

```
cpChar:  MACRO
           LDA  \1
           STA  \2
         ENDM
```

Listing 8.36 Example assembler output listing

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
    1    1                              XDEF Start
    2    2                     MyData:  SECTION
    3    3   000000            char1:   DS.B  1
    4    4   000001            char2:   DS.B  1
    5    5                              INCLUDE "macro.inc"
    6   1i                     cpChar:  MACRO
    7   2i                              LDA  \1
    8   3i                              STA  \2
    9   4i                              ENDM
   10    6                     CodeSec: SECTION
   11    7                     Start:
   12    8                              cpChar char1, char2
   15    9   000006 9D                  NOP
```

The Assembler stores the content of included files in the listing file. The Assembler also stores the macro definition and invocation in the listing file.

The Assembler does not store the macro expansion lines in the listing file.

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

-L: Generate a listing file

-Lc: No Macro call in listing file

-Ld: No macro definition in listing file-Li: No included file in listing file

-Li: No included file in listing file

Group

Output

Scope

Assembly unit

Syntax

`-Li`

Arguments

None

Default

None

Description

Switches on the generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definition, invocation, and expansion lines.

Example

`ASMOPTIONS=-Li`

In the following example of assembly code, the `cpChar` macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When `-Li` option is specified, the assembly source code in Listing 8.37 along with additional source code (Listing 8.38) from the `macro.inc` file generates the following output in the assembly listing file:

Listing 8.37 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 8.38 Example source code in an include file

```
cpChar:  MACRO
         LDA  \1
         STA  \2
         ENDM
```

Listing 8.39 Example assembler output listing

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF  Start
    2    2                      MyData:  SECTION
    3    3   000000             char1:   DS.B  1
    4    4   000001             char2:   DS.B  1
    5    5                               INCLUDE "macro.inc"
   10    6                      CodeSec: SECTION
   11    7                      Start:
   12    8                               cpChar char1, char2
   13   2m   000000 C6  xxxx   +         LDA  char1
   14   3m   000003 C7  xxxx   +         STA  char2
   15    9   000006 9D                   NOP
```

The Assembler stores the macro definition, invocation, and expansion in the listing file.

The Assembler does not store the content of included files in the listing file.

For a detailed description of the listing file, see the Assembler Listing File chapter.

See also

Assembler options:

- -L: Generate a listing file
- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file

-Lic: License information

Group

Various

Scope

None

Syntax

```
-Lic
```

Arguments

None

Default

None

Description

The `-Lic` option prints the current license information (e.g., if it is a demo version or a full version). This information is also displayed in the *About* box.

Example

```
ASMOPTIONS=-Lic
```

See also

Assembler options:

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

-LicA: License information about every feature in directory

Group

Various

Scope

None

Syntax

```
-LicA
```

Arguments

None

Default

None

Description

The `-LicA` option prints the license information of every tool or DLL in the directory where the executable is (e.g., if tool or feature is a demo version or a full version). Because the option has to analyze every single file in the directory, this may take a long time.

Example

```
ASMOPTIONS=-LicA
```

See also

Assembler options:

- -Lic: License information
- -LicBorrow: Borrow license feature
- -LicWait: Wait until floating license is available from floating License Server

-LicBorrow: Borrow license feature

Group

Host

Scope

None

Syntax

`-LicBorrow<feature>[;<version>]:<Date>`

Arguments

`<feature>`: the feature name to be borrowed (e.g., `HI100100`).

`<version>`: optional version of the feature to be borrowed (e.g., `3.000`).

`<date>`: date with optional time until when the feature shall be borrowed (e.g., `15-Mar-2005:18:35`).

Default

None

Defines

None

Pragmas

None

Description

This option lets you borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool is aware of the version). However, if you want to borrow any feature, you need to specify the feature's version number.

You can check the status of currently borrowed features in the tool's *About* box.

NOTE You only can borrow features if you have a floating license and if your floating license is enabled for borrowing. See the provided FLEXlm documentation about details on borrowing.

Example

```
-LicBorrowHI100100;3.000:12-Mar-2005:18:25
```

See also

Assembler options:

- -Lic: License information
- -LicA: License information about every feature in directory
- -LicWait: Wait until floating license is available from floating License Server

-LicWait: Wait until floating license is available from floating License Server

Group

Host

Scope

None

Syntax

```
-LicWait
```

Arguments

None

Default

None

Description

If a license is not available from the floating license server, then the default condition is that the application will immediately return. With the `-LicWait` assembler option set, the application will wait (blocking) until a license is available from the floating license server.

Example

```
ASMOPTIONS=-LicWait
```

See also

Assembler options:

- -Lic: License information
- -LicA: License information about every feature in directory
- -LicBorrow: Borrow license feature

-Ll: Show label statistics

Group

Output

Syntax

```
-Ll
```

Arguments

None

Description

It displays label statistics in the list file.The option gives the gain in terms of code size for a label if moved to SHORT or TINY section.

Example:

```
Test.asm:
TINY_RAM_VARS: SECTION  RS08_SHORT;Insert your data
```

```
definition here
tmp:        DS.B   1..
FiboLoop:   STA    tmp ; store last
RTS
Test.lst:
Freescale Assembler
 Ind.  Name             tiny       short
 ----  ----             ----       -----
    1  tmp                      1           1
```

-M (-Ms, -Mt): Memory model

Group

Code Generation

Scope

Application

Syntax

`-M(s|b|t)`

Arguments

`s`: small memory model

`t`: tiny memory model

Default

`-Ms`

Description

The Assembler for the MC68HC(S)08 supports two different memory models. The default is the small memory model, which corresponds to the normal setup, i.e., a 64kB code-address space. The tiny memory model corresponds to the situation where the default RAM is in the zero page.

NOTE For the Assembler, the memory model does not matter at all. The memory model is used by the compiler to specify the default allocation of variable and functions. The Assembler has this option only to generate “compatible” object files for the memory model consistency check of the linker.

NOTE In the tiny memory model, the default for the compiler is to use zero-page addressing. The default for the Assembler is to still use extended-addressing modes. See the Using the direct addressing mode to access symbols section to see how to generate zero-page accesses.

Example

```
ASMOPTIONS=-Mt
```

-MacroNest: Configure maximum macro nesting

Group

Language

Scope

Assembly Unit

Syntax

```
-MacroNest<Value>
```

Arguments

`<Value>`: max. allowed nesting level

Default

```
3000
```

Description

This option controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations.

Example

See the description of message A1004 for an example.

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 the MCUasm Compatibility chapter in the Appendices.

Example

```
ASMOPTIONS=-MCUasm
```

-MMU: Enable Memory Management Unit (MMU) Support

Group

CODE GENERATION

Scope

Assembly Unit

Syntax

`-MMU`

Arguments

None

Default

None

Defines

`__MMU__`

Pragmas

None

Description

This option enables code banking and CALL and RTC instructions are available. It can be used only when `-Cs08` is enabled.

-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 chapter

-ObjN: Object filename specification

Group

Output

Scope

Assembly Unit

Syntax

`-ObjN<FileName>`

Arguments

`<FileName>`: Name of the binary output file generated.

Default

`-ObjN%n.o` when generating a relocatable file or

`-ObjN%n.abs` when generating an absolute file.

Description

Normally, the object file has the same name than the processed source file, but with the `.o` extension when relocatable code is generated or the `.abs` extension when absolute code is generated. This option allows a flexible way to define the output filename. The modifier `%n` can also be used. It is replaced with the source filename. If `<file>` in this option contains a path (absolute or relative), the `OBJPATH` environment variable is ignored.

Example

For `ASMOPTIONS=-ObjNa.out`, the resulting object file will be `a.out`. If the `OBJPATH` environment variable is set to `\src\obj`, the object file will be `\src\obj\a.out`.

For `fibo.c -ObjN%n.obj`, the resulting object file will be `fibo.obj`.

For `myfile.c -ObjN..\objects\_%n.obj`, the object file will be named relative to the current directory to `...\objects\_myfile.obj`. Note that the environment variable `OBJPATH` is ignored, because `<file>` contains a path.

See also

OBJPATH: Object file path environment variable

-Prod: Specify project file at startup

Group

None (This option cannot be specified interactively.)

Scope

None

Syntax

```
-Prod=<file>
```

Arguments

`<file>`: name of a project or project directory

Default

None

Description

This option can only be specified at the command line while starting the application. It cannot be specified in any other circumstances, including the `default.env` file, the command line or whatever.

When this option is given, the application opens the file as configuration file. When the filename does only contain a directory, the default name `project.ini` is appended. When the loading fails, a message box appears.

Example

```
assembler.exe -Prod=project.ini
```

(Use the Assembler's executable name instead of `assembler`.)

See also

Environment chapter

-Struct: Support for structured types

Group

Input

Scope

Assembly Unit

Syntax

```
-Struct
```

Arguments

None

Default

None

Description

When this option is activated, the Macro Assembler also support the definition and usage of structured types. This is interesting for application containing both ANSI-C and Assembly modules.

Example

```
ASMOPTIONS=-Struct
```

See also

Mixed C and Assembler Applications chapter

-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 8.40):

Listing 8.40 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 the following:

- "`Window`": Application window has the default window size.
- "`Min`": Application window is minimized.
- "`Max`": Application window is maximized.
- "`Hidden`": Application window is not visible (only if there are arguments).

Default

Application is started with arguments: `Minimized`.

Application is started without arguments: `Window`.

Description

Normally, the application is started with a normal window if no arguments are given. If the application is started with arguments (e.g., from the Maker to assemble, compile, or link a file), then the application is running minimized to allow for batch processing. However, the application's window behavior may be specified with the View option.

Using `-ViewWindow`, the application is visible with its normal window. Using `-ViewMin` the application is visible iconified (in the task bar). Using `-ViewMax`, the application is visible maximized (filling the whole screen). Using `-ViewHidden`, the application processes arguments (e.g., files to be compiled or linked) completely invisible in the background (no window or icon visible in the task bar). However, for example, if you are using the -N: Display notify box assembler option, a dialog box is still possible.

Example

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

-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

-WOutFile: Create error listing file

-Wmsg8x3: Cut filenames in Microsoft format to 8.3

Group

Messages

Scope

Assembly Unit

Syntax

`-Wmsg8x3`

Default

None

Description

Some editors (e.g., early versions of WinEdit) are expecting the filename in the Microsoft message format in a strict 8.3 format. That means the filename can have at most 8 characters with not more than a 3-character extension. Using a newer Windows OS, longer file names are possible. With this option the filename in the Microsoft message is truncated to the 8.3 format.

Example

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

With the -Wmsg8x3 option set, the above message will be

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

See also

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
- -WmsgFoi: Message format for interactive mode
- -WmsgFob: Message format for batch mode Option
- --WmsgFonp: Message format for no position information

-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 8.41 is to be assembled in the batch mode.

Listing 8.41 Example assembly source code

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

The Assembler generates the error output (Listing 8.42) in the assembler window if it is running in batch mode:

Listing 8.42 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 8.43 Example error listing in the verbose format for batch mode

```
ASMOPTIONS=-WmsgFbv
>> in "C:\tw2.asm", line 6, col 0, pos 81
  endif
^
ERROR A1001: Conditional else not allowed here
```

See also

ERRORFILE: Filename specification error

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

-WmsgFob: Message format for batch mode

-WmsgFoi: Message format for interactive mode

-WmsgFonf: Message format for no file information

-WmsgFonp: Message format for no position information

-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 8.45.

Listing 8.44 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 8.45 Example error listing in Microsoft format for interactive mode

```
ASMOPTIONS=-WmsgFim
X:\TWE.ASM(12): ERROR: conditional else not allowed here
```

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFob: Message format for batch mode
- -WmsgFoi: Message format for interactive mode
- -WmsgFonf: Message format for no file information
- -WmsgFonp: Message format for no position information

-WmsgFob: Message format for batch mode

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgFob<string>`

Arguments

`<string>`: format string (see below).

Default

```
-WmsgFob"%f%e(%l): %K %d: %m\n"
```

Description

With this option it is possible to modify the default message format in the batch mode. The formats in Listing 8.46 are supported (assumed that the source file is `x:\Freescale\sourcefile.asmx`).

Listing 8.46 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 8.47, using the format in Listing 8.46. The options are set for producing the path of a file with its filename, extension, and line.

Listing 8.47 Error message

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

See also

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
- -WmsgFoi: Message format for interactive mode
- -WmsgFonf: Message format for no file information
- -WmsgFonp: Message format for no position information

-WmsgFoi: Message format for interactive mode

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFoi<string>
```

Arguments

`<string>`: format string (see below)

Default

```
-WmsgFoi"\n>> in \"%f%e\", line %l, col %c, pos
%o\n%s\n%K %d: %m\n"
```

Description

With this option it is possible modify the default message format in interactive mode. The following formats are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`):

Listing 8.48 Supported message formats - interactive mode

```
Format Description           Example
-----------------------------------------------------
%s     Source Extract
```

```
%p      Path                  x:\Freescale\
%f      Path and name         x:\Freescale\sourcefile
%n      Filename              sourcefile
%e      Extension             .asmx
%N      File (8 chars)        sourcefi
%E      Extension (3 chars)   .asm
%l      Line                  3
%c      Column                47
%o      Pos                   1234
%K      Uppercase kind        ERROR
%k      Lowercase kind        error
%d      Number                A1051
%m      Message               text
%%      Percent               %
\n      New line
```

Example

`ASMOPTIONS=-WmsgFoi"%f%e(%l): %k %d: %m\n"`

produces a message in following format (Listing 8.49):

Listing 8.49 Error message resulting from the statement above

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

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
- -WmsgFob: Message format for batch mode
- -WmsgFonf: Message format for no file information
- -WmsgFonp: Message format for no position information

-WmsgFonf: Message format for no file information

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgFonf<string>`

Arguments

`<string>`: format string (see below)

Default

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

Description

Sometimes there is no file information available for a message (e.g., if a message not related to a specific file). Then this message format string is used. The following formats are supported:

```
Format Description         Example
----------------------------------
%K     Uppercase kind      ERROR
%k     Lowercase kind      error
%d     Number              L10324
%m     Message             text
%%     Percent             %
\n     New line
```

Example

`ASMOPTIONS=-WmsgFonf”%k %d: %m\n”`

produces a message in following format:

`information L10324: Linking successful`

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
- -WmsgFob: Message format for batch mode
- -WmsgFoi: Message format for interactive mode
- -WmsgFonp: Message format for no position information

-WmsgFonp: Message format for no position information

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFonp<string>
```

Arguments

`<string>`: format string (see below)

Default

```
-WmsgFonp"%f%e: %K %d: %m\n"
```

Description

Sometimes there is no position information available for a message (e.g., if a message not related to a certain position). Then this message format string is used. The following formats are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`)

Listing 8.50 Supported message formats for when there is no position information

```
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 environment variable

Assembler options:

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
- -WmsgFob: Message format for batch mode
- -WmsgFoi: Message format for interactive mode
- -WmsgFonf: Message format for no file information

-WmsgNe: Number of error messages

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgNe<number>`

Arguments

`<number>`: Maximum number of error messages.

Default

`50`

Description

With this option the amount of error messages can be reported until the Assembler stops assembling. Note that subsequent error messages which depends on a previous one may be confusing.

Example

`ASMOPTIONS=-WmsgNe2`

The Assembler stops assembling after two error messages.

See also

Assembler options:

- -WmsgNi: Number of Information messages
- -WmsgNw: Number of Warning messages

-WmsgNi: Number of Information messages

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgNi<number>`

Arguments

`<number>`: Maximum number of information messages.

Default

`50`

Description

With this option the maximum number of information messages can be set.

Example

```
ASMOPTIONS=-WmsgNi10
```

Only ten information messages are logged.

See also

Assembler options:

- -WmsgNe: Number of error messages
- -WmsgNw: Number of Warning messages

-WmsgNu: Disable user messages

Group

Messages

Scope

None

Syntax

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

Arguments

`a`: Disable messages about include files

`b`: Disable messages about reading files

`c`: Disable messages about generated files

`d`: Disable messages about processing statistics

`e`: Disable informal messages

Default

None

Description

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

- `a`: The application provides information about all included files. With this suboption this option can be disabled.
- `b`: With this suboption messages about reading files e.g., the files used as input can be disabled.
- `c`: Disables messages informing about generated files.
- `d`: At the end of the assembly, the application may provide information about statistics, e.g., code size, RAM/ROM usage, and so on. With this suboption this option can be disabled.
- `e`: With this option, informal messages (e.g., memory model, floating point format, etc.) can be disabled.

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

Example

```
-WmsgNu=c
```

-WmsgNw: Number of Warning messages

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgNw<number>
```

Arguments

`<number>`: Maximum number of warning messages.

Default

`50`

Description

With this option the maximum number of warning messages can be set.

Example

`ASMOPTIONS=-WmsgNw15`

Only 15 warning messages are logged.

See also

Assembler options:

- -WmsgNe: Number of error messages
- -WmsgNi: Number of Information messages

-WmsgSd: Setting a message to disable

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgSd<number>`

Arguments

`<number>`: Message number to be disabled, e.g., `1801`

Default

None

Description

With this option a message can be disabled so it does not appear in the error output.

Example

```
-WmsgSd1801
```

See also

Assembler options:

- -WmsgSe: Setting a message to Error
- -WmsgSi: Setting a message to Information
- -WmsgSw: Setting a Message to Warning

-WmsgSe: Setting a message to Error

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgSe<number>
```

Arguments

`<number>`: Message number to be an error, e.g., `1853`

Default

None

Description

Allows changing a message to an error message.

Example

```
-WmsgSe1853
```

See also

- -WmsgSd: Setting a message to disable
- -WmsgSi: Setting a message to Information
- -WmsgSw: Setting a Message to Warning

-WmsgSi: Setting a message to Information

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgSi<number>`

Arguments

<number>: Message number to be an information, e.g., `1853`

Default

None

Description

With this option a message can be set to an information message.

Example

`-WmsgSi1853`

See also

Assembler options:

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to Error
- -WmsgSw: Setting a Message to Warning

-WmsgSw: Setting a Message to Warning

Group

Messages

Scope

Assembly Unit

Syntax

`-WmsgSw<number>`

Arguments

`<number>`: Error number to be a warning, e.g., `2901`

Default

None

Description

With this option a message can be set to a warning message.

Example

`-WmsgSw2901`

See also

Assembler options:

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to Error
- -WmsgSi: Setting a message to Information

-WOutFile: Create error listing file

Group

Messages

Scope

Assembly Unit

Syntax

`-WOutFile(On|Off)`

Arguments

None

Default

Error listing file is created.

Description

This option controls if a error listing file should be created at all. The error listing file contains a list of all messages and errors which are created during a assembly process. Since the text error feedback can now also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the listing file is controlled by the environment variable ERRORFILE: Filename specification error.

Example

`-WOutFileOn`

The error file is created as specified with `ERRORFILE`.

`-WErrFileOff`

No error file is created.

See also

Assembler options:

- -WErrFile: Create "err.log" error file
- -WStdout: Write to standard output

-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
- -WOutFile: Create error listing file

9

Sections

Sections are portions of code or data that cannot be split into smaller elements. Each section has a name, a type, and some attributes.

Each assembly source file contains at least one section. The number of sections in an assembly source file is only limited by the amount of memory available on the system at assembly time. If several sections with the same name are detected inside of a single source file, the code is concatenated into one large section.

Sections from different modules, but with the same name, will be combined into a single section at linking time.

Sections are defined through Section attributes and Section types. The last part of the chapter deals with the merits of using relocatable sections. (See Relocatable vs. absolute sections.)

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
- Relocatable sections

Absolute sections

The starting address of an absolute section is known at assembly time. An absolute section is defined through the ORG - Set Location Counter assembler directive. The operand specified in the `ORG` directive determines the start address of the absolute section. See Listing 9.1 for an example of constructing absolute sections using the `ORG` assembler directive.

Listing 9.1 Example source code using ORG for absolute sections

```
       XDEF  entry
       ORG   $8000 ; Absolute constant data section.
cst1:  DC.B  $26
cst2:  DC.B  $BC
...
       ORG   $080  ; Absolute data section.
var:   DS.B  1
       ORG   $8010 ; Absolute code section.
entry:
```

```
      LDA   cst1  ; Loads value in cst1
      ADD   cst2  ; Adds value in cst2
      STA   var   ; Stores result into var
      BRA   entry
```

In the previous example, two bytes of storage are allocated starting at address `$A00`. The *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 9.2.

Listing 9.2 Example PRM file for Listing 9.1

```
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 assembler directive. See Listing 9.3 for an example using the `SECTION` directive.

Listing 9.3 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:
          LDA    cst1  ; Load value into cst1
          ADD    cst2  ; Add value in cst2
          STA    var   ; Store into var
          BRA    entry
```

In the previous example, two bytes of storage are allocated in the `constSec` section. The constant `cst1` is allocated at the start of the section at address `$A00` and another constant `cst2` is allocated at an offset of `1` byte from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable `constSec` section until another section is specified using the `ORG` or `SECTION` directives.

When using relocatable sections, the user does not need to care about overlapping sections. The linker will assign a start address to each section according to the input from the linker parameter file.

The user can decide to define only one memory area for the code and constant sections and another one for the variable sections or to split the sections over several memory areas.

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 9.4.

Listing 9.4 PRM file for Listing 9.3 defining one RAM area and one ROM area

```
LINK  test.abs/* Name of the executable file generated.     */
NAMES test.o  /* Name of the object file in the application */
END

SECTIONS
/* READ_ONLY memory area.               */
  MY_ROM  = READ_ONLY  0x8000 TO 0xFDFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x0100 TO 0x023F;
END

PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.          */
  DEFAULT_RAM, dataSec  , SSTACK INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM, constSec          INTO MY_ROM;
END
INIT     entry /* entry is the entry point to the application. */
VECTOR ADDRESS 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 files which should be linked (`NAMES` command).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined `DEFAULT_RAM` section (or its ELF alias `.data`) must be placed there (`SECTIONS` and `PLACEMENT` commands).
- The specification of a memory area where the sections containing code or constants must be allocated. At least, the predefined `DEFAULT_ROM` section (or its ELF alias `.text`) must be placed there (`SECTIONS` and `PLACEMENT` commands).

- Constants sections should be defined in the ROM memory area in the `PLACEMENT` section (otherwise, they are allocated in RAM).
- The specification of the application entry point (`INIT` command).
- The definition of the reset vector (`VECTOR ADDRESS` command).

According to the PRM file above:

- the `dataSec` section will be allocated starting at `0x0080`.
- the `codeSec` section will be allocated starting at `0x0B00`.
- the `constSec` section will be allocated next to the `codeSec` section.

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 9.5:

Listing 9.5 PRM file for Listing 9.3 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;
  MY_STK    = READ_WRITE 0x0200 TO 0x023F; /* Stack memory area     */
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec              INTO RAM_AREA_2;
  DEFAULT_RAM          INTO RAM_AREA_1;
  SSTACK               INTO MY_STK; /* Stack allocated in MY_STK   */

/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec             INTO ROM_AREA_2;
  codeSec, DEFAULT_ROM INTO ROM_AREA_1;
END
INIT   entry                /* Application's entry point.          */
VECTOR 0 entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

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

According to the PRM file in Listing 9.5,

- 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.

10

Assembler Syntax

An assembler source program is a sequence of source statements. Each source statement is coded on one line of text and can be either a:

- Comment line or a
- Source line.

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 10.1). 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 10.1 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,
- an Operation field,
- one or several operands, or
- a comment.

Characters on the source line may be either upper or lower case. Directives and instructions are case-insensitive, whereas symbols are case-sensitive unless the assembler option for case insensitivity on label names (-Ci: Switch case sensitivity on label names OFF) is activated.

Label field

The label field is the first field in a source line. A label is a symbol followed by a colon. Labels can include letters (A–Z or a–z), underscores, periods and numbers. The first character must not be a number.

NOTE For compatibility with other Assembler vendors, an identifier starting on column 1 is considered to be a label, even when it is not terminated by a colon. When the -MCUasm: Switch compatibility with MCUasm ON assembler option is activated, you *MUST* terminate labels with a colon. The Assembler produces an error message when a label is not followed by a colon.

Labels are required on assembler directives that define the value of a symbol (`SET` or `EQU`). For these directives, labels are assigned the value corresponding to the expression in the operand field.

Labels specified in front of another directive, instruction or comment are assigned the value of the location counter in the current section.

NOTE When the Macro Assembler expands a macro it generates internal symbols starting with an underscore `'_'`. Therefore, to avoid potential conflicts, user defined symbols should not begin with an underscore

NOTE For the Macro Assembler, a `.B` or `.W` at the end of a label has a specific meaning. Therefore, to avoid potential conflicts, user- defined symbols should not end with `.B` or `.W`.

Operation field

The operation field follows the label field and is separated from it by a white space. The operation field must not begin in the first column. An entry in the operation field is one of the following:

- an instruction's mnemonic - an abbreviated, case-insensitive name for a member in the Instruction set
- a Directive name, or
- a Macro name.

Instruction set

Executable instructions for the M68HC08 processor are defined in the *CPU08 Reference Manual.*

HC08 instruction set

Table 10.1 presents an overview of the instructions available for the HC08:

Table 10.1 HC08 instruction set

Instruction	Addressing modes	Descriptions
ADC	#<expression> <expression> <expression>,X ,X <expression>,SP	Add with Carry
ADD	#<expression> <expression> <expression>,X ,X <expression>,SP	Add without carry
AIS	#<expression>	Add Immediate value (signed) to Stack Pointer
AIX	#<expression>	Add Immediate value (signed) to Index register H:X
AND	#<expression> <expression> <expression>,X ,X <expression>,SP	Logical AND
ASL	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Left
ASLA		Arithmetic Shift Left Accumulator
ASLX		Arithmetic Shift Left register X

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
ASR	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Right
ASRA		Arithmetic Shift Right Accumulator
ASRX		Arithmetic Shift Right register X
BCC	<label>	Branch if Carry bit Clear
BCLR	BitNumber, <expression>	Clear one Bit in memory
BCS	<label>	Branch if Carry bit Set
BEQ	<label>	Branch if Equal
BGE	<label>	Branch if Greater Than or Equal to
BGND		Enter Background Debug Mode. Only available for HCS08 (-CS08 option)
BGT	<label>	Branch if Greater Than
BHCC	<label>	Branch if Half Carry bit Clear
BHCS	<label>	Branch if Half Carry bit Set
BHI	<label>	Branch if Higher
BHS	<label>	Branch if Higher or Same
BIH	<label>	Branch if /IRQ Pin High
BIL	<label>	Branch if /IRQ Pin Low
BIT	#<expression> <expression> <expression>,X ,X <expression>,SP	Bit Test
BLE	<label>	Branch if Less Than or Equal To

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
BLO	<label>	Branch if Lower (same as BCS)
BLS	<label>	Branch if Lower or Same
BLT	<label>	Branch if Less Than
BMC	<label>	Branch if interrupt Mask Clear
BMI	<label>	Branch if Minus
BMS	<label>	Branch If interrupt Mask Set
BNE	<label>	Branch if Not Equal
BPL	<label>	Branch if Plus
BRA	<label>	Branch Always
BRCLR	BitNumber, <expression>, <label>	Branch if Bit is Clear
BRN	<label>	Branch Never
BRSET	BitNumber, <expression>, <label>	Branch if Bit Set
BSET	BitNumber,<expression>	Set Bit in memory
BSR	<label>	Branch to Subroutine
CBEQ	<expression>,<label> <expression>,X+,<label> X+,<label> <expression>,SP,<label>	Compare and Branch if Equal
CBEQA	#<expression>,<label>	
CBEQX	#<expression>,<label>	
CLC		Clear Carry bit
CLI		Clear Interrupt mask bit

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
CLR	<expression> <expression>,X ,X <expression>,SP	Clear memory
CLRA		Clear Accumulator A
CLRH		Clear index Register H
CLRX		Clear index Register X
CMP	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare accumulator with memory
COM	<expression> <expression>,X ,X <expression>,SP	One's complement on memory location
COMA		One's complement on accumulator A
COMX		One's complement on register X
CPHX	#<expression> <expression> <expression>,SP	Compare index register H:X with memory Stack pointer and Extended addressing modes only available for HCS08 (-CS08 option)
CPX	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare index register X with memory
DAA		Decimal Adjust Accumulator

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
DBNZ	<expression>,<label> <expression>,X,<label> X,<label> <expression>,SP,<label>	Decrement counter and Branch if Not Zero
DBNZA	<label>	
DBNZX	<label>	
DEC	<expression> <expression>,X ,X <expression>,SP	Decrement memory location
DECA		Decrement Accumulator
DECX		Decrement Index register
DIV		Divide
EOR	#<expression> <expression> <expression>,X ,X <expression>,SP	Exclusive OR Memory with accumulator
INC	<expression> ,X <expression>,X <expression>,SP	Increment memory location
INCA		Increment Accumulator
INCX		Increment register X
JMP	<expression> <expression>,X ,X	Jump to label

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
JSR	<expression> <expression>,X ,X	Jump to Subroutine
LDA	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Accumulator
LDHX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Index register H:X from memory Indexed, Stack pointer and extended addressing modes are only available for HCS08 (-CS08 option).
LDX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load index Register X from memory
LSL	<expression> <expression>,X ,X <expression>,SP	Logical Shift Left in memory
LSLA		Logical Shift Left Accumulator
LSLX		Logical Shift Left register X

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
LSR	<expression> <expression>,X ,X <expression>,SP	Logical Shift Right in memory
LSRA		Logical Shift Right Accumulator
LSRX		Logical Shift Right register X
MOV	<expression>,<expression> <expression>,X+ #<expression>,<expression> X+,<expression>	Memory-to-memory byte Move
MUL		Unsigned multiply
NEG	<expression> <expression>,X ,X <expression>,SP	Two’s complement in memory
NEGA		Two’s complement on Accumulator
NEGX		Two’s complement on register X
NOP		No operation
NSA		Nibble Swap Accumulator
ORA	#<expression> <expression> <expression>,X ,X <expression>,SP	Inclusive OR between Accumulator and memory
PSHA		Push Accumulator onto stack
PSHH		Push index register H onto stack
PSHX		Push index register X onto stack

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
PULA		Pull Accumulator from stack
PULH		Pull index register H from stack
PULX		Pull index register X from stack
ROL	<expression> <expression>,X ,X <expression>,SP	Rotate memory Left
ROLA		Rotate Accumulator Left
ROLX		Rotate register X Left
ROR	<expression> <expression>,X ,X <expression>,SP	Rotate memory Right
RORA		Rotate Accumulator Right
RORX		Rotate register X Right
RSP		Reset Stack Pointer
RTI		Return from Interrupt
RTS		Return from Subroutine
SBC	#<expression> <expression> <expression>,X ,X <expression>,SP	Subtract with Carry
SEC		Set Carry bit
SEI		Set Interrupt mask bit

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
STA	<expression> <expression>,X ,X <expression>,SP	Store Accumulator in Memory
STHX	<expression> <expression>,SP	Store Index register H:X Stack pointer and extended addressing modes are only available for HCS08 (-CS08 option)
STOP		Enable IRQ pin and Stop oscillator
STX	<expression> <expression>,X ,X <expression>,SP	Store index register X in memory
SUB	#<expression> <expression> <expression>,X ,X <expression>,SP	Subtract
SWI		Software Interrupt
TAP		Transfer Accumulator to CCR
TAX		Transfer Accumulator to index Register X
TPA		Transfer CCR to Accumulator
TST	<expression> <expression>,X ,X <expression>,SP	Test memory for negative or zero
TSTA		Test Accumulator for negative or zero

Table 10.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Descriptions
TSTX		Test register X for negative or zero
TSX		Transfer SP to index register H:X
TXA		Transfer index register X to Accumulator
TXS		Transfer index register X to SP
WAIT		Enable interrupts; stop processor

Special HCS08 instructions

The HCS08 core provides the following instructions in addition to the HC08 core instructions (Table 10.2):

Table 10.2 Special HC(S)08 instructions

Instruction	Addressing modes	Descriptions
BGND		Enter Background Debug Mode. Only available with the -CS08/-C08/-CRS08: Derivative family assembler options.
CPHX	#<expression> <expression> <expression>,SP	Compare index register H:X with memory Stack pointer and extended addressing modes are only available with the -CS08, -C08, or -CRS08 assembler options.
LDHX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load index register H:X from memory Indexed, stack pointer, and extended addressing modes are only available with the -CS08 option
STHX	<expression> <expression>,SP	Store index register H:X Stack pointer and extended addressing modes are only available with the -CS08 option.

RS08 instruction set

Table 10.3 presents an overview of the instructions available for the RS08.

Table 10.3 RS08 instructions set

Instruction	Addressing Modes	Description
ADC	#<expression> <expression> ,X D[X] X	Add with Carry
ADCX		Alias for ADC X
ADD	#<expression> <expression> ,X D[X] X	Add without Carry
ADDX		Alias for ADD X
AND	#<expression> <expression> ,X D[X] X	Logical AND
ANDX		Alias for AND X
ASLA		Arithmetic Shift Left Accumulator (alias for LSLA)
BCC	<label>	Branch if Carry Bit Clear
BCLR	BitNumber, <expression> BitNumber,D[X] BitNumber,X	Clear one Bit in Memory
BCS	<label>	Branch if Carry Bit Set
BEQ	<label>	Branch if Equal
BGND		Background
BHS	<label>	Branch if Higher or Same
BLO	<label>	Branch if Lower

Table 10.3 RS08 instructions set (*continued*)

Instruction	Addressing Modes	Description
BNE	<label>	Branch if Not Equal
BRN	<label>	Branch Never (Alias for BRA *+$2)
BRCLR	BitNumber, <expression>, <label> BitNumber,D[X],<label> BitNumber,X,<label>	Branch if Bit is Clear
BRSET	BitNumber, <expression>, <label> BitNumber,D[X],<label> BitNumber,X,<label>	Branch if Bit Set
BSET	BitNumber,<expression> BitNumber,D[X] BitNumber,X	Set Bit in Memory
BSR	<label>	Branch to Subroutine
CBEQ	<expression>,<label> #<expression>,<label> ,X,<label> D[X],<label> X,<label>	Compare and Branch if Equal
CBEQA	<label>	
CBEQX	<label>	
CLC		Clear Carry Bit
CLR	<expression> ,X D[X] X	Clear Memory
CLRX		Clear Index Register X
CMP	#<expression> <expression> ,X D[X] X	Compare Accumulator with Memory
COMA		Complement (One’s Complement)

Table 10.3 RS08 instructions set (*continued*)

Instruction	Addressing Modes	Description
DBNZ	<expression>,<label> ,X,<label> D[X],<label> X,<label>	Decrement Counter and Branch if Not Zero
DBNZA	<label>	
DBNZX	<label>	
DEC	<expression> ,X D[X] X	Decrement Memory Location
DEC	<$13	Force tiny addressing (will use $03)
DECA		Decrement Accumulator
DECX		Decrement Index Register
EOR	#<expression> <expression> D[X] ,X X	Exclusive OR Memory with Accumulator
EORX		Exclusive OR (index register and accumulator)
INC	<expression> ,X D[X] X	Increment Memory Location
INC	>$01	Force direct addressing
INCA		Increment Accumulator
INCX		Increment Register X
JMP	<label>	Jump to Label
JSR	<label>	Jump to Subroutine

Table 10.3 RS08 instructions set (*continued*)

Instruction	Addressing Modes	Description
LDA	#<expression> <expression> ,X D[X] X	Load Accumulator indexed
LDA	<$0FF	Force short addressing (will use $1F)
LDX	#<expression> <expression> ,X D[X] X	Load Index Register X from Memory
LDX	$OFF	Load Direct
LSLA		Logical Shift Left Accumulator
LSRA		Logical Shift Right Accumulator
MOV	<expression>,<expression> #<expression>,<expression> D[X],<expression> <expression>,D[X] #<expression>,D[X]	Memory to Memory Byte Move
NOP		No Operation
ORA	#<expression> <expression> ,X D[X] X	Inclusive OR between Accumulator and Memory
ORAX		Inclusive OR between Accumulator and Index Register
ROLA		Rotate Accumulator Left
RORA		Rotate Accumulator Right
RTS		Return from Subroutine

Table 10.3 RS08 instructions set (*continued*)

Instruction	Addressing Modes	Description
SBC	#<expression> <expression> ,X D[X] X	Subtract with Carry
SBCX		Subtract with Carry (Index Register content from Accumulator)
SEC		Set Carry Bit
SHA		Swap Shadow PC High with A
SLA		Swap Shadow PC Low with A
STA	<expression> ,X D[X] X	Store Accumulator in Memory
STOP		Stop Processing
STX	<expression>	Store Index Register X in Memory
SUB	#<expression> <expression> ,X D[X]	Subtract
SUBX		
TAX		Transfer Accumulator to Index Register X
TST	#<expression> <expression> ,X D[X]	Test for zero (alias for MOV <expression>,<expression>)
TSTA		Test Accumulator (alias for ORA #0)
TSTX		Test Index Register X (alias for MOV X,X)
TXA		Transfer Index Register X to Accumulator
WAIT		Enable Interrupts; Stop Processor

NOTE For RS08 both D[X] and ,X notations refer to the memory location $000E. The ,X notation is supported for compatibility reasons with HC(S)08. Wherever ,X is supported, D[X] is also supported. In situations where the use of ,X would lead to double commas (e.g. BCLR 0,,X) the use of ,X is not allowed.

Directive

Assembler directives are described in the Assembler Directives 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 chapter in this manual.

Operand field: Addressing modes (HC(S)08)

The operand fields, when present, follow the operation field and are separated from it by a white space. When two or more operand subfields appear within a statement, a comma must separate them.

The following addressing mode notations are allowed in the operand field (Table 10.4):

Table 10.4 HC(S)08 addressing mode notation

Addressing Mode	Notation	Example
Inherent	No operands	RSP
Immediate	#<expression>	ADC #$01
Direct	<expression>	ADC byte
Extended	<expression>	ADC word
Indexed, no offset	,X	ADC ,X
Indexed, 8-bit offset	<expression>,X	ADC Offset,X
Indexed, 16-bit offset	<expression>,X	ADC Offset,X
Relative	<label>	BRA Label
Stack Pointer, 8-bit offset	<expression>,SP	ADC Offset,SP

Table 10.4 HC(S)08 addressing mode notation (*continued*)

Addressing Mode	Notation	Example
Stack Pointer, 16-bit offset	<expression>,SP	ADC Offset,SP
Memory-to-memory immediate-to-direct	#<expression>,<expression>	MOV #$05,MyDataByte
Memory-to-memory direct-to-direct	<expression>,<expression>	MOV DatLoc1,DatLoc2
Memory-to-memory indexed-to-direct with post-increment	X+,<expression>	MOV X+,<expression>
Memory-to-memory direct-to-indexed with post-increment	<expression>,X+	MOV <expression>,X+
Indexed with post-increment	X+	CBEQ X+, Data
Indexed, 8-bit offset, with post-increment	#<expression>,X+	CBEQ #offset,X+,Data

Inherent

Instructions using this addressing mode do not have any associated instruction fetch (Listing 10.2). Some of them are acting on data in the CPU registers.

Listing 10.2 Inherent addressing-mode instructions

```
    CLRA
    DAA
```

Immediate

The opcode contains the value to use with the instruction rather than the address of this value.

The effective address of the instruction is specified using the # character as in Listing 10.3.

Listing 10.3 Immediate addressing mode

```
           XDEF   Entry
```

```
initStack: EQU   $0400

MyData:    SECTION
data:      DS.B  1

MyCode:    SECTION
Entry:
           LDHX  #initStack ; init Stack Pointer
           TXS              ; with value $400-1 = $03FF

main:      LDA   #100       ; load register A with (decimal) 100
           BRA   main
```

In this example, the hexadecimal value `$0400` is loaded in value in the register HX and the decimal value `100` is loaded into register A.

Direct

The direct addressing mode is used to address operands in the direct page of the memory (location `$0000` to `$00FF`).

For most of the direct instructions, only two bytes are required: the first byte is the opcode and the second byte is the operand address located in page zero. See Listing 10.4 for an example of the direct addressing mode.

Listing 10.4 Direct addressing mode

```
           XDEF  Entry
initStack: EQU   $0400
MyData:    SECTION SHORT
data:      DS.B  1
MyCode:    SECTION
Entry:
           LDHX  #initStack ; init Stack Pointer
           TXS              ; with value $400 - 1 = $03FF
main:      LDA   #$55
           STA   data
           BRA   main
```

In this example, the value `$55` is stored in the variable data, which is located on the direct page. The `MyData` section must be defined in the direct page in the linker parameter file. The opcode generated for the `STA data` instruction is two bytes long.

Extended

The extended addressing mode is used to access memory location located above the direct page in a 64-kilobyte memory map.

For the extended instructions, three bytes are required: the first byte is the opcode and the second and the third bytes are the most and least significant bytes of the operand address. See Listing 10.5 for an example of the extended addressing mode.

Listing 10.5 Extended addressing mode

```
           XDEF  Entry
initStack: EQU   $0400
           ORG   $B00
data:      DS.B  1
MyCode:    SECTION
Entry:
           LDHX  #initStack ; init Stack Pointer
           TXS              ; with value $400-1 = $03FF
main:      LDA   #$55
           STA   data
           BRA   main
```

In this example, the value `$55` is stored in the variable data. This variable is located at address `$0B00` in the memory map. The opcode of the `STA data` instruction is then three bytes long.

Indexed, no offset

This addressing mode is used to access data with variable addresses through the HX index register of the HC08 controller. The X index register contains the least significant byte of the operand while the H index register contains the most significant byte.

Indexed, no offset instructions are one byte long. See Listing 10.6 for an example of using the indexed (no offset) addressing mode.

Listing 10.6 Indexed (no offset) addressing mode

```
...
Entry:
  ...
  LDHX  #$0FFE
  LDA   ,X
  ...
  JMP   ,X
  ...
```

The value stored in memory location `$0FFE` is loaded into accumulator A. The `JMP` instruction causes the program to jump to the address pointed to by the HX register.

Indexed, 8-bit offset

This addressing mode is useful when selecting the k-th element in an n-element table. The size of the table is limited to 256 bytes.

Indexed, 8-bit offset instructions are two byte long. The first byte is the opcode and the second byte contains the index register offset byte. See Listing 10.7 for an example of using the indexed (8-bit offset) addressing mode.

Listing 10.7 Index (8-bit offset) addressing mode

```
            XDEF   Entry
initStack:  EQU    $0400
MyData:     SECTION SHORT
data:       DS.B   8
MyCode:     SECTION
Entry:
            LDHX   #initStack ; init Stack Pointer
            TXS               ; with value $400-1 = $03FF
main:
            LDHX   #data
            LDA    5 ,X
            ...
            JMP    $FF,X
   ...
```

The value contained in the memory at the location calculated using the address of data (pointed to by the HX register) + 5 is loaded in accumulator A. The `JMP` instruction causes the program to jump to the address pointed to by the HX register + `$FF`.

Indexed, 16-bit offset

This addressing mode is useful when selecting the k-th element in an n-element table. The size of the table is limited to `$FFFF` bytes.

Indexed,16-bit offset instructions are three byte long. The first byte contains the opcode and the second and the third the high and low index register offset bytes. See Listing 10.8 for an example of using the indexed (16-bit offset) addressing mode.

Listing 10.8 Indexed (16-bit offset) addressing mode

```
            XDEF  Entry
initStack: EQU   $0400
MyData:     SECTION
data:       DS.B  8
MyCode:     SECTION
Entry:
            LDHX  #initStack ; init Stack Pointer
            TXS              ; with value $400-1 = $03FF
main:
            LDHX  #table
            STA   $500 ,X
            ...
            JMP   $1000,X
  ...
```

The value contained in the memory at the location calculated using the address of data (pointed to by register `HX`) + $500 is loaded in accumulator `A`. The `JMP` instruction causes the program to jump to the address pointed to by the `HX` register + `$1000`.

Relative

This addressing mode is used by all branch instructions to determine the destination address. The signed byte following the opcode is added to the contents of the program counter.

As the offset is coded on a signed byte, the branching range is -127 to +128. The destination address of the branch instruction must be in this range. See Listing 10.9 for an example of using the relative addressing mode.

Listing 10.9 Relative addressing mode

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

Stack Pointer, 8-bit offset

Stack Pointer, 8-bit offset instructions behave the same way than Indexed 8-bit offset instructions, except that the offset is added to the Stack Pointer SP in place of the HX Index register.

This addressing mode allow easy access of the data on the stack. If the interrupts are disabled, the Stack pointer can also be used as a second Index register. See Listing 10.10 for an example of using the Stack Pointer *8-bit offset) addressing mode.

Listing 10.10 Stack Pointer (8-bit offset) addressing mode

```
entry:
       LDHX   #$0500    ; init Stack Pointer to 04FF
       TXS

       LDA    #$40
       STA    $50, SP    ; Location $54F = $40
```

In this example, stack pointer, 8-bit offset mode is used to store the value `$40` in memory location `$54F`.

Stack Pointer, 16-bit offset

Stack Pointer, 16-bit offset instructions behave the same way than Indexed, 16-bit offset instructions, except that the offset is added to the Stack Pointer (SP) in place of the HX Index register.

This addressing mode allow easy access of the data on the stack. If the interrupts are disabled, the Stack pointer can also be used as a second Index register. See Listing 10.11 for an example of using the Stack Pointer (16-bit offset) addressing mode.

Listing 10.11 Stack Pointer (16-bit offset) addressing mode

```
entry:
       LDHX   #$0100     ; init Stack Pointer to 00FF
       TXS

       LDA    $0500, SP ; Content of memory location $5FF is loaded in A
```

In this example, stack pointer, 16-bit offset mode is used to store the value in memory location $5FF in accumulator A.

Memory-to-memory immediate-to-direct

This addressing mode is generally used to initialize variables and registers in page zero. The register `A` is not affected. See Listing 10.12 for an example for using the memory-to-memory immediate-to-direct addressing mode.

Listing 10.12 Memory-to-memory immediate-to-direct addressing mode

```
MyData:   EQU   $50
entry:
          MOV   #$20, MyData
```

The `MOV #$20,MyData` instruction stores the value `$20` in memory location `$50` `‘MyData’`.

Memory-to-memory direct-to-direct

This addressing mode is generally used to transfer variables and registers in page zero. The A register is not affected. See Listing 10.13 for an example of using the memory-to-memory direct-to-direct addressing mode.

Listing 10.13 Memory-to-memory direct-to-direct addressing mode

```
MyData1:  EQU   $50
MyData2:  EQU   $51
entry:
          MOV   #$10, MyData1
          MOV   MyData1, MyData2
```

The `MOV #$10,MyData1` instruction stores the value `$10` in memory location `$50` `‘MyData1’` using the memory-to-memory Immediate-to-Direct addressing mode. The `MOV MyData1,MyData2` instruction moves the content of `MyData1` into `MyData2` using memory to memory Direct-to-Direct addressing mode. The content of `MyData2` (memory location `$51`) is then `$10`.

Memory-to-memory indexed-to-direct with post-increment

This addressing mode is generally used to transfer tables addressed by the index register to a register in page zero.

The operand addressed by the HX index register is stored in the direct page location addressed by the byte following the opcode. The HX index register is automatically incremented. The A register is not affected. See Listing 10.14 for an example of using the memory-to-memory indexed to direct with post-increment addressing mode.

Listing 10.14 Memory-to-memory indexed-to-direct with post increment addressing mode.

```
            XDEF   Entry

ConstSCT:   SECTION
Const:      DC.B   1,11,21,31,192,12,0

DataSCT:    SECTION SHORT
MyReg:      DS.B   1

CodeSCT:    SECTION
Entry:      LDHX   #$00FF
            TXS
main:
            LDHX   #Const
LOOP:       MOV    X+, MyReg
            BEQ    main
            BRA    LOOP
```

In this example, the table `Const` contains seven bytes defined in a constant section in ROM. The last value of this table is zero.

The HX register is initialized with the address of `Const`. All the values of this table are stored one after another in page-zero memory location `MyReg` using the `MOV X+, MyReg` instruction. When the value `0` is encountered, the HX register is reset with the address of the first element of the `#Const` table.

Memory-to-memory direct-to-indexed with post-increment

This addressing mode is generally used to fill tables addressed by the index register from registers in page zero.

The operand in the direct page location addressed by the byte following the opcode is stored in the memory location pointed to by the HX index register. The HX index register is automatically incremented. The A register is not affected. See Listing 10.15 for an example of using the memory-to-memory direct-to-indexed with post-increment addressing mode.

Listing 10.15 Memory-to-memory direct-to-indirect with post-increment addressing mode

```
            XDEF   entry
MyData:     SECTION SHORT
```

```
MyReg1:   DS.B   1
MyReg2:   DS.B   1
MyCode:   SECTION
entry:
          LDA    #$02
          STA    MyReg1
          INCA
          STA    MyReg2

          LDHX    #$1000
          MOV    MyReg1,X+
          MOV    MyReg2,X+
main:     BRA    main
```

The page-zero memory locations `MyReg1` and `MyReg2` are first respectively initialized with `$02` and `$03`. The contents of those data are then written in memory location `$1000` and `$1001`. The HX register points to memory location `$1002`.

Indexed with post-increment

The operand is addressed then the `HX` register is incremented.

This addressing mode is useful for searches in tables. It is only used with the `CBEQ` instruction. See Listing 10.16 for an example of an example of using the indexed with post-increment addressing mode.

Listing 10.16 Example of the indexed with post-increment addressing mode

```
          XDEF  Entry
          ORG   $F000
data:     DC.B  1,11,21,31,$C0,12
CodeSCT:  SECTION
Entry:    LDHX  #$00FF
          TXS
main:
          LDA   #$C0
          LDHX  #data
LOOP:     CBEQ  X+,IS_EQUAL

          BRA   LOOP
IS_EQUAL: ...
```

Using this addressing mode, it is possible to scan the memory to find a location containing a specific value.

The value located at the memory location pointed to by HX is compared to the value in the A register. If the two values match, the program branches to IS_EQUAL. HX points to the memory location next to the one containing the searched value.

In this example, the value $C0 is searched starting at memory location $F000. This value is found at the memory location $F004, the program branches to IS_EQUAL, and the HX register contains $F005.

Indexed, 8-bit offset, with post-increment

The address of the operand is the sum of the 8-bit offset added to the value in register HX.

The operand is addressed, then the HX register is incremented.

This addressing mode is useful for searches in tables. It is only used with the CBEQ instruction. See Listing 10.17 for an example of the indexed (8-bit offset) with post-increment addressing mode.

Listing 10.17 Indexed (8-bit offset) with post-increment addressing mode

```
          XDEF   Entry
          ORG    $F000
data:     DCB.B  $40,$00
          DC.B   1,11,21,31,$C0,12 ; $C0 is located at $F000+$40+4
CodeSCT:  SECTION
Entry:    LDHX   #$00FF
          TXS
main:
          LDA    #$C0
          LDHX   #data
LOOP:     CBEQ   $30,X+,IS_EQUAL
          BRA    LOOP
IS_EQUAL: ...
```

Using this addressing mode, it is possible to scan the memory to find a location containing a specific value starting at a specified location to which is added an offset.

The value located at memory location pointed to by HX + $30 is compared to the value in the A register. If the two values match, program branch to IS_EQUAL. HX points to memory location next to the one containing the searched value.

In this example, the value $C0 is searched starting at memory location $F000+$30=$F030. This value is found at memory location $F044, the program branches to IS_EQUAL. The HX register contains the memory location of the searched value minus the offset, incremented by one: $F044-$30+1=$F015.

Operand Field: Addressing Modes (RS08)

The following addressing mode notations are allowed in the operand field for the RS08:

Table 10.5 Operand Field RS08 Addressing Modes

Inherent	No operands	RTS
Tiny	<expression>	ADD fourbits
Short	<expression>	CLR fivebits
Direct	<expression>	ADC byte
Extended	<expression>	JSR word
Relative	<label>	BRA Label
Immediate	#<expression>	ADC #$01
Indexed	D[X] or ,X	ADC D[X] or ADC ,X

Inherent (RS08)

Instructions using this addressing mode have no associated instruction fetch. Some of them are acting on data in the CPU registers.

Example:

```
   CLRA
   INCA
   NOP
```

Tiny

The tiny addressing mode is used to access only the first 16 bytes of the memory map (addresses from `$0000` to `$000F`). The instructions using this addressing mode are encoded using one byte only. This addressing mode is available for `INC`, `DEC`, `ADD` and `SUB` instructions.

Example:

```
           XDEF Entry
MyData:    SECTION RS08_TINY
data:      DS.B 1
MyCode:    SECTION
Entry:
```

```
main:      ADD data
           BRA main
```

In this example, the value of the variable data is added to the accumulator. The data is located in the tiny memory area, so the encoding of the ADD instruction will be one byte long. Note that the tiny section has to be placed into the tiny memory area at link time.

Short

The RS08 short addressing mode is used to access only the first 32 bytes of the memory map (addresses from $0000 to $001F). The instructions using this addressing mode are encoded using one byte only. This addressing mode is available for CLR, LDA and STA instructions.

Example:

```
           XDEF Entry
MyData:    SECTION RS08_SHORT
data:      DS.B 1
MyCode:    SECTION
Entry:
main:      LDA data
           BRA main
```

In this example, the value of the variable data is loaded into the accumulator. The data is located in the short memory area, so the encoding of the LDA instruction will be one byte long. Note that the short section has to be placed into the tiny memory area at linktime.

Direct

The direct addressing mode is used to address operands in the direct page of the memory (location $0000 to $00FF).

Example:

```
           XDEF Entry
MyData:    SECTION
data:      DS.B 1
MyCode:    SECTION
Entry:
main:      LDA #$55
           STA data
           BRA main
```

In this example, the value `$55` is stored in the variable data. The opcode generated for the instruction `STA data` is two bytes long.

Extended

The extended addressing mode is used only for `JSR` and `JMP` instructions. The 14-bit address is located in the lowest 14 bits of the encoding after the two-bit opcode.

Example:

```
           XDEF Entry
           XREF target
data:      DS.B 1
MyCode:    SECTION
Entry:
main:      LDA #$55
           JMP target
```

In this example a jump is executed at an address defined by the external symbol target.

Relative

This addressing mode is used by all branch instructions to determine the destination address. The signed byte following the opcode is added to the contents of the program counter.

As the offset is coded on a signed byte, the branching range is -127 to +128. The destination address of the branch instruction must be in this range.

Example:

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

Immediate

The opcode contains the value to use with the instruction rather than the address of this value. The effective address of the instruction is specified using the # character as in the example below.

Example:

```
           XDEF Entry
MyData:    SECTION
```

```
data:       DS.B 1

MyCode:     SECTION
Entry:
main:       LDA #100
            BRA main
```

In this example, the decimal value 100 is loaded in register `A`.

Indexed

When using the indexed addressing mode, an index register is used as reference to access the instruction's operand. For the RS08, the index registers are located at `$000F` (register X) and `$000E` (register D[X]). The D[X] register is called the index data register, and can be designated by either one of the D[X] or `,X` notations. As a restriction, when the use of `,X` would lead to double commas in the assembly source, the use of `,X` is not allowed.

Example:

```
            XDEF Entry
MyData:     SECTION
data:       DS.B 1

MyCode:     SECTION
Entry:
main:       CLR D[X] ; equivalent to CLR ,X
            CLR X
```

In this example the contents of both X and D[X] registers are replaced by zeros.

Comment Field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in the comment field.

Example:

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

Symbols

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

- User-defined symbols
- External symbols
- Undefined symbols
- Reserved symbols

User-defined symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

- The section, in which the memory location is defined
- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION - Declare Relocatable Section assembler directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line (Listing 10.18).

Listing 10.18 Example of a user-defined relocatable SECTION

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec.
label2: DC.B 5 ; label2 is assigned offset 2 within Sec.
label3: DC.B 1 ; label3 is assigned offset 7 within Sec.
```

It is also possible to define a label with either an absolute or a previously defined relocatable value, using the SET - Set Symbol Value or EQU - Equate symbol value assembler directives.

Symbols with absolute values must be defined with constant expressions.

Listing 10.19 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 assembler directive. In another source file, an XREF - External Symbol Reference assembler directive must reference it. Since its address is unknown in the referencing file, it is considered to be relocatable. See Listing 10.20 for an example of using `XDEF` and `XREF`.

Listing 10.20 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 10.21 shows an example of an undeclared label.

Listing 10.21 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 HC08 processor the reserved identifiers are listed in Listing 10.22.

Listing 10.22 Reserved identifiers for an HC(S)08 derivative

```
A, CCR, H, X, SP
```

The keywords `LOW` and `HIGH` are also reserved identifiers. They are used to refer to the low byte and the high byte of a memory location.

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 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 10.23 for a variety of string constants.

Listing 10.23 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)
- Multiplication, division and modulo operators (binary)
- Sign operators (unary)
- Shift operators (binary)
- Bitwise operators (binary)
- Logical operators (unary)
- Relational operators (binary)
- HIGH operator
- PAGE operator
- Force operator (unary)

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 10.24 for an example of addition and subtraction operators.

Listing 10.24 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 10.25 for an example of the multiplication, division, and modulo operators.

Listing 10.25 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 10.26 for an example of the unary sign operators.

Listing 10.26 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 10.27 for an example of the binary shift operators.

Listing 10.27 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 10.28 for an example of the binary bitwise operators

Listing 10.28 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 10.29 for an example of the unary bitwise operator.

Listing 10.29 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 10.30 for an example of the unary logical operator.

Listing 10.30 Unary logical operator

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

Relational operators (binary)

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

Syntax

```
Equal:                 <operand> =  <operand>
                       <operand> == <operand>
Not equal:             <operand> != <operand>
                       <operand> <> <operand>
Less than:             <operand> <  <operand>
Less than or equal:    <operand> <= <operand>
Greater than:          <operand> >  <operand>
Greater than or equal: <operand> >= <operand>
```

Description

These operators compare two operands and return `1` if the condition is true or `0` if the condition is false.

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

Example

See Listing 10.31 for an example of the binary relational operators

Listing 10.31 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.

`LDA #HIGH(data1)`

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

`LDA HIGH(data1)`

This instruction will load the direct value at memory location of the higher byte of the address of `data1` (i.e., the value in memory location `$10`) in register A.

HIGH_6_13 Operator

Syntax

```
High Byte:  HIGH_6_13(<operand>)
```

Description

This operator returns the high byte of a 14-bit address of a memory location.

Example

Assume `data1` is a word located at address `$1010` in the memory.

```
LDA   #HIGH_6_13(data1)
```

This instruction will load the value $40 in the accumulator.

LOW operator

The LOW operator is `LOW`.

Syntax

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

Description

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

Example

Assume `data1` is a word located at address `$1050` in the memory.

```
LDA #LOW(data1)
```

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

```
LDA LOW(data1)
```

This instruction will load the direct value at memory location of the lower byte of the address of `data1` (i.e., the value in memory location `$50`) in register A.

MAP_ADDR_6 Operator

Syntax

```
MAP_ADDR_6(<operand>)
```

Description

This operator returns the lower 6 bits for a memory location. It should be used to determine the offset in the paging window for a certain memory address.Note that the operator automatically adds the offset of the base of the paging window ($C0).

Example

```
MOV    #HIGH_6_13(data), $001F
STA    MAP_ADDR_6(data)
```

In this example, the RS08 PAGE register (mapped at $001F) is loaded with the memory page corresponding to data and then the value contained in the accumulator is stored at the address pointed by data.

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.

```
LDA #PAGE(data1)
```

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

```
LDA PAGE(data1)
```

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

NOTE The PAGE keyword does not refer to the RS08 PAGE register but to the PAGE operator described above.

Force operator (unary)

Syntax

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

Description

The < or `.B` operators force direct addressing mode, whereas the > or `.W` operators force extended addressing mode.

Use the < operator to force 8-bit indexed or 8-bit direct addressing mode for an instruction.

Use the > operator to force 16-bit indexed or 16-bit extended addressing mode for an instruction.

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

Example

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

For the RS08 the < operand forces the operand to short or tiny addressing mode (depending on the instruction in which it is used). The same result can be obtained by adding `.S` or `.T` to the referred symbol. The > operator forces an address to 8 bits, even if it fits in 4 or 5 bits (so short or tiny addressing modes can be used).

Operator precedence

Operator precedence follows the rules for ANSI - C operators (Table 10.6)

Table 10.6 Operator precedence priorities

Operator	Description	Associativity
()	Parenthesis	Right to Left
~ + -	One’s complement Unary Plus Unary minus	Left to Right
* / %	Integer multiplication Integer division Integer modulo	Left to Right
+ -	Integer addition Integer subtraction	Left to Right
<< >>	Shift Left Shift Right	Left to Right
< <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to Right
=, == !=, <>	Equal to Not Equal to	Left to Right
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
|	Bitwise OR	Left to Right

Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols
- External symbols
- The special symbol '*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the `DC` directive:

  `DC.W  1, 2, *-2`

Once a valid expression has been fully evaluated by the Assembler, it is reduced as one of the following type of expressions:

- Absolute expression: The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant.
  Simple relocatable expression: The expression evaluates to an absolute offset from the start of a single relocatable section.
- Complex relocatable expression: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The Assembler does not support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in `XREF` directives, which are assumed to be relocatable symbols.

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 10.32 for an example of an absolute expression.

Listing 10.32 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 10.33 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 10.34):

Listing 10.34 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 10.35, 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 10.35 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 10.36 Example of relocatable expression

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

Unary operation result

Table 10.7 describes the type of an expression according to the operator in an unary operation:

Table 10.7 Expression type resulting from operator and operand type

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

Binary operations result

Table 10.8 describes the type of an expression according to the left and right operators in a binary operation:

Table 10.8 Expression type resulting from operator and their operands

Operator	Left Operand	Right Operand	Expression
-	absolute	absolute	absolute
-	relocatable	absolute	relocatable
-	absolute	relocatable	complex
-	relocatable	relocatable	absolute
+	absolute	absolute	absolute
+	relocatable	absolute	relocatable
+	absolute	relocatable	relocatable
+	relocatable	relocatable	complex
*, /, %, <<, >>, |, &, ^	absolute	absolute	absolute
*, /, %, <<, >>, |, &, ^	relocatable	absolute	complex
*, /, %, <<, >>, |, &, ^	absolute	relocatable	complex
*, /, %, <<, >>, |, &, ^	relocatable	relocatable	complex

Translation limits

The following limitations apply to the Macro Assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Include may be nested up to `50`.
- The maximum line length is `1023`.

11

Assembler Directives

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

Directive overview

Section-Definition directives

Use the directives in Table 11.1 to define new sections.

Table 11.1 Directives for defining sections

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

Constant-Definition directives

Use the directives in Table 11.2 to define assembly constants.

Table 11.2 Directives for defining constants

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

Data-Allocation directives

Use the directives in Table 11.3 to allocate variables.

Table 11.3 Directives for allocating variables

Directive	Description
DC - Define Constant	Define a constant variable
DCB - Define Constant Block	Define a constant block
DS - Define Space	Define storage for a variable
RAD50 - RAD50-encoded string constants	RAD50 encoded string constants

Symbol-Linkage directives

Symbol-linkage directives (Table 11.4) are used to export or import global symbols.

Table 11.4 Symbol linkage directives

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

Assembly-Control directives

Assembly-control directives (Table 11.5) are general purpose directives used to control the assembly process.

Table 11.5 Assembly control directives

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

Listing-File Control directives

Listing-file control directives (Table 11.6) control the generation of the assembler listing file.

Table 11.6 Listing-file control directives

Directive	Description
CLIST - List conditional assembly	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable Listing	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set Line Length	Define line length in assembly listing file.
MLIST - List macro expansions	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable Listing	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable Paging	Disable paging in the assembly listing file.
PAGE - Insert Page break	Insert page break.
PLEN - Set Page Length	Define page length in the assembler listing file.
SPC - Insert Blank Lines	Insert an empty line in the assembly listing file.
TABS - Set Tab Length	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide Listing Title	Define the user defined title for the assembler listing file.

Macro Control directives

Macro control directives (Table 11.7) are used for the definition and expansion of macros.

Table 11.7 Macro control directives

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

Conditional Assembly directives

Conditional assembly directives (Table 11.8) are used for conditional assembling.

Table 11.8 Conditional assembly directives

Directive	Description
ELSE - Conditional assembly	alternate block
ENDIF - End conditional assembly	End of conditional block
IF - Conditional assembly	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater than or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less than or equal to 0.
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 11.1 is assembled using the `-FA2` assembler option, an ELF/DWARF 2.0 Absolute file is generated.

Listing 11.1 Using ABSENTRY to specify an application entry point

```
       ABSENTRY entry

       ORG   $fffe
Reset: DC.W  entry
       ORG   $70
entry: NOP
       NOP
```

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

According to the ABSENTRY directive, the entry point will be set to the address of entry in the header of the absolute file.

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 11.2 aligns the HEX label to a location, which is a multiple of 16 (in this case, location 00010 (Hex))

Listing 11.2 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 11.3 for examples of setting the number base.

Listing 11.3 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 assembler option is activated, the Assembler defaults to `CLIST ON`.

Example

Listing 11.4 is an example where the `CLIST OFF` option is used.

Listing 11.4 Listing file with CLIST OFF

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

Listing 11.5 is the corresponding listing file.

Listing 11.5 Example assembler listing where CLIST ON is used

```
Abs. Rel.   Loc    Obj. code   Source line
---- -----  ------ ---------   -----------
   2     2         0000 0000   Try: EQU   0
   3     3         0000 0000        IFEQ  Try
   4     4  000000 A667               LDA   #103
   5     5                          ELSE
   7     7                          ENDIF
```

Listing 11.6 is a listing file using `CLIST ON`.

Listing 11.6 CLIST ON is selected

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

Listing 11.7 is the corresponding listing file.

Listing 11.7 Example assembler listing where CLIST ON is used

```
Abs. Rel.   Loc    Obj. code   Source line
---- -----  ------ ---------   -----------
   2     2         0000 0000   Try: EQU   0
   3     3         0000 0000        IFEQ  Try
   4     4  000000 A667               LDA   #103
   5     5                          ELSE
   6     6                            LDA   #0
   7     7                          ENDIF
   8     8
```

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 11.8).
- `DC.W`: Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary (Listing 11.9).
- `DC.L`: Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary (Listing 11.10).

Listing 11.8 Example for DC.B

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

Listing 11.9 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 11.10 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 assembler truncates the value and generates a warning message.

See also

Assembler directives:

- DCB - Define Constant Block
- DS - Define Space
- ORG - Set Location Counter
- SECTION - Declare Relocatable Section

DCB - Define Constant Block

Syntax

```
[<label>:]  DCB [.<size>] <count>, <value>
```

`where <size>` = `B (default)`, `W`, or `L`.

Description

The `DCB` directive causes the Assembler to allocate a memory block initialized with the specified `<value>`. The length of the block is `<size> * <count>`.

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

The value of each storage unit allocated is the sign-extended expression `<value>`, which may contain forward references. The `<count>` cannot be relocatable. This directive does not perform any alignment.

The following rules apply to size specifications for `DCB` directives:

- `DCB.B`: One byte is allocated for numeric expressions.
- `DCB.W`: Two bytes are allocated for numeric expressions.
- `DCB.L`: Four bytes are allocated for numeric expressions.

Listing 11.11 Examples of DCB directives

```
000000 FFFF FF      Label: DCB.B 3, $FF
000003 FFFE FFFE           DCB.W 3, $FFFE
000007 FFFE
000009 0000 FFFE           DCB.L 3, $FFFE
00000D 0000 FFFE
000011 0000 FFFE
```

See also

Assembler directives:

- DC - Define Constant
- DS - Define Space
- ORG - Set Location Counter
- SECTION - Declare Relocatable Section

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 11.12). The content of the memory reserved is not initialized. The length of the block is <size> * <count>.

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

Listing 11.12 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 11.13, 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 11.14).

Listing 11.13 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 11.14 Proper memory allocation

```
DataSect:      SECTION    ; separate section for variables
Counter:       DS 1       ; 1-byte used

ConstSect:     SECTION   ; separate section for constants
InitialCounter: DC.B $f5  ; constant $f5

CodeSect:      SECTION    ; section for code
main:           NOP       ; NOP instruction
```

An `ORG` directive also starts a new section.

See also

- DC - Define Constant
- ORG - Set Location Counter
- SECTION - Declare Relocatable Section

ELSE - Conditional assembly

Syntax

```
IF <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

`ELSEC`

Description

If `<condition>` is true, the statements between `IF` and the corresponding `ELSE` directive are assembled (generate code).

If `<condition>` is false, the statements between `ELSE` and the corresponding `ENDIF` directive are assembled. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Example

Listing 11.15 is an example of the use of conditional assembly directives:

Listing 11.15 Various conditional assembly directives

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

The value of `Try` determines the instruction to be assembled in the program. As shown, the `lda #103` instruction is assembled. Changing the operand of the `EQU` directive to `0` causes the `lda #0` instruction to be assembled instead.

Listing 11.16 Output listing of Listing 11.15

```
 Abs.  Rel.     Loc     Obj. code    Source line
---- ----   ------ ---------   -----------------
    1     1          0000 0001   Try: EQU   1
    2     2          0000 0001        IF  Try != 0
    3     3   000000 A667               LDA   #103
    4     4                           ELSE
    6     6                           ENDIF
```

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 11.17 causes any source code after the END statement to be ignored, as in Listing 11.18.

Listing 11.17 Source File

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

Listing 11.18 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 11.28 in the FOR section.

See also

Assembler directives:

- FOR - Repeat assembly block
- -Compat: Compatibility modes

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 11.30 in the `IF` section.

See also

IF - Conditional assembly assembler directive

ENDM - End macro definition

Syntax

`ENDM`

Synonym

None

Description

The `ENDM` directive terminates the macro definition (Listing 11.19).

Example

The ENDM statement in Listing 11.19 terminates the cpChar macro.

Listing 11.19 Using ENDM to terminate a macro definition

```
cpChar:  MACRO
           LDA   \1
           STA   \2
         ENDM
CodeSec: SECTION
Start:
         cpChar char1, char2
         LDA   char1
         STA   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 11.20 for examples of using the `EQU` directive.

Listing 11.20 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 11.21 for instances where the EVEN directive causes padding bytes to be inserted.

Listing 11.21 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 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 11.22 generates the error messages in Listing 11.23. 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 11.22 Example source code

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

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

Listing 11.23 Error messages resulting from assembling the source code in Listing 11.22

```
>> 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 11.24 is another assembly code example which again incorporates the `FAIL 200` and the `FAIL 600` directives. Listing 11.25 is the error message that was generated as a result of assembling the source code in Listing 11.24.

Listing 11.24 Example source code

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

          IFC "\2", ""
            FAIL  600
          ELSE
            STA  \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
         cpChar, char2
```

Listing 11.25 Error messages resulting from assembling the source code in Listing 11.24

```
>> 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 11.26 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 11.27.

Listing 11.26 Example source code

```
cpChar: MACRO
          IFC "\1", ""
            FAIL  "A character must be specified as first parameter"
            MEXIT
          ELSE
            LDA  \1
          ENDIF

          IFC "\2", ""
            FAIL  600
          ELSE
            STA  \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
        cpChar, char2
```

Listing 11.27 Error messages resulting from assembling the source code in Listing 11.26

```
>> 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 11.28 is an example of using `FOR` to create a 5-repetition loop.

Listing 11.28 Using the FOR directive in a loop

```
FOR label=2 TO 6
  DC.B  label*7
ENDFOR
```

Listing 11.29 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

ENDFOR - End of FOR block

-Compat: Compatibility modes assembler option

IF - Conditional assembly

Syntax

```
IF <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

None

Description

If `<condition>` is true, the statements immediately following the `IF` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached. Then all the statements until the corresponding `ENDIF` directive are ignored. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

The expected syntax for `<condition>` is:

```
<condition> := <expression> <relation> <expression>
<relation>  := =|!=|>=|>|<=|<|<>
```

The <expression> must be absolute (It must be known at assembly time).

Example

Listing 11.30 is an example of the use of conditional assembly directives

Listing 11.30 IF and ENDIF

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

The value of Try determines the instruction to be assembled in the program. As shown, the `lda    #0` instruction is assembled. Changing the operand of the `EQU` directive to one causes the `lda    #103` instruction to be assembled instead. The following shows the listing provided by the Assembler for these lines of code:

Listing 11.31 Output listing after conditional assembly

```
    1    1           0000 0000    Try: EQU   0
    2    2           0000 0000         IF Try != 0
    4    4                             ELSE
    5    5    000000 A600              LDA   #0
    6    6                             ENDIF
```

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 11.9 lists the available conditional types:

Table 11.9 Conditional assembly types

Ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Example

Listing 11.32 is an example of the use of conditional assembler directives:

Listing 11.32 Using the IFNE conditional assembler directive

```
Try: EQU    0
     IFNE   Try
       LDA    #103
     ELSE
       LDA    #0
     ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `lda    #0` instruction is assembled. Changing the directive to `IFEQ` causes the `lda    #103` instruction to be assembled instead.

Listing 11.33 shows the listing provided by the Assembler for these lines of code

Listing 11.33 output listing for Listing 11.32

```
   1    1            0000 0000   Try: EQU   0
   2    2            0000 0000   IFNE  Try
   4    4                        ELSE
   5    5    000000 A600           LDA   #0
   6    6                        ENDIF
```

INCLUDE - Include text from another file

Syntax

```
INCLUDE <file specification>
```

Synonym

None

Description

This directive causes the included file to be inserted in the source input stream. The `<file specification>` is not case-sensitive and must be enclosed in quotation marks.

The Assembler attempts to open `<file specification>` relative to the current working directory. If the file is not found there, then it is searched for relative to each path specified in the GENPATH: Search path for input file environment variable.

Example

```
INCLUDE "..\LIBRARY\macros.inc"
```

LIST - Enable Listing

Syntax

```
LIST
```

Synonym

None

Description

Specifies that instructions following this directive must be inserted into the listing and into the debug file. This is a default option. The listing file is only generated if the -L: Generate a listing file assembler option is specified on the command line.

The source text following the `LIST` directive is listed until a NOLIST - Disable Listing or an END - End assembly assembler directive is reached

This directive is not written to the listing and debug files.

Example

The assembly source code using the `LIST` and `NOLIST` directives in Listing 11.34 generates the output listing in Listing 11.35.

Listing 11.34 Using the LIST and NOLIST assembler directives

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP          NOP
```

Listing 11.35 Output listing generated from running Listing 11.34

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
   1    1    000000 9D         aaa:   NOP
   2    2
   4    4    000001 9D         bbb:   NOP
   5    5    000002 9D                NOP
   6    6
  12   12    000005 9D         ddd:   NOP
  13   13    000006 9D                NOP
```

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 11.37 generates the listing file in Listing 11.37. Notice that the `LLEN 24` directive causes the output at the location-counter line 7 to be truncated.

Listing 11.36 Example assembly source code using LLEN

```
        DC.B   $55
        LLEN   32
        DC.W   $1234, $4567
```

```
        LLEN  24
        DC.W  $1234, $4567
        EVEN
```

Listing 11.37 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 11.38 for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

Listing 11.38 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
```

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 11.39 for a macro definition.

Listing 11.39 Example macro definition

```
         XDEF  Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
cpChar:  MACRO
           LDA   \1
           STA   \2
         ENDM
CodeSec: SECTION
Start:
         cpChar char1, char2
```

```
         LDA     char1
         STA     char2
```

MEXIT - Terminate Macro Expansion

Syntax

```
MEXIT
```

Synonym

None

Description

`MEXIT` is usually used together with conditional assembly within a macro. In that case it may happen that the macro expansion should terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the ENDM - End macro definition directive.

Example

See Listing 11.40 allows the replication of simple instructions or directives using `MACRO` with `MEXIT`.

Listing 11.40 Example assembly code using MEXIT

```
         XDEF   entry

storage: EQU    $00FF
save:    MACRO            ; Start macro definition
         LDX    #storage
         LDA    \1
         STA    0,x       ; Save first argument
         LDA    \2
         STA    2,x       ; Save second argument
         IFC    '\3', ''  ; Is there a third argument?
           MEXIT          ; No, exit from macro
         ENDC
         LDA    \3        ; Save third argument
           STA    4,X
         ENDM             ; End of macro definition
datSec:  SECTION
char1:  ds.b   1
```

```
char2:  ds.b  1
codSec:  SECTION
entry:
         save  char1, char2
```

Listing 11.41 shows the macro expansion of the previous macro.

Listing 11.41 Macro expansion of Listing 11.40

```
Abs. Rel.   Loc    Obj. code  Source line
---- ----   ------ ---------  -----------
   1    1                               XDEF  entry
   2    2          0000 00FF   storage: EQU   $00FF
   3    3
   4    4                      save:    MACRO ; Start macro definition
   5    5                               LDX   #storage
   6    6                               LDA   \1
   7    7                               STA   0,x ; Save first arg
   8    8                               LDA   \2
   9    9                               STA   2,x ; Save second arg
  10   10                               IFC   '\3', ''; is there a
  11   11                               MEXIT ; No, exit from macro.
  12   12                               ENDC
  13   13                               LDA   \3 ; Save third argument
  14   14                               STA   4,X
  15   15                               ENDM     ; End of macro defin
  16   16
  17   17                       datSec:  SECTION
  18   18    000000             char1:  ds.b  1
  19   19    000001             char2:  ds.b  1
  20   20
  21   21
  22   22
  23   23                       codSec:  SECTION
  24   24                       entry:
  25   25                                save  char1, char2
  26    5m  000000 AEFF       +          LDX   #storage
  27    6m  000002 C6 xxxx    +          LDA   char1
  28    7m  000005 E700       +          STA   0,x  ; Save first arg
  29    8m  000007 C6 xxxx    +          LDA   char2
  30    9m  00000A E702       +          STA   2,x  ; Save second
  31   10m         0000 0001  +          IFC   '', ''  ; Is there a
  33   11m                    +          MEXIT      ; no, exit macro.
  34   12m                    +          ENDC
  35   13m                    +          LDA        ; Save third argu
  36   14m                    +          STA   4,X
```

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 11.42, with `MLIST ON`, generates the assembler output listing in Listing 11.43

Listing 11.42 Example assembly source code using MLIST

```
        XDEF   entry
        MLIST ON
swap:   MACRO
          LDA   \1
          LDX   \2
          STA   \2
          STX   \1
       ENDM
codSec: SECTION
entry:
        LDA   #$F0
        LDX   #$0F
main:
        STA   first
        STX   second
        swap  first, second
        NOP
        BRA   main
datSec: SECTION
```

```
first:  DS.B   1
second: DS.B   1
```

Listing 11.43 Assembler output listing of the example in Listing 11.42 with MLIST ON

```
   1    1                             XDEF  entry
   3    3                     swap:   MACRO
   4    4                               LDA   \1
   5    5                               LDX   \2
   6    6                               STA   \2
   7    7                               STX   \1
   8    8                             ENDM
   9   9
  10   10                     codSec: SECTION
  11   11                     entry:
  12   12   000000 A6F0               LDA   #$F0
  13   13   000002 AE0F               LDX   #$0F
  14   14                     main:
  15   15   000004 C7 xxxx            STA   first
  16   16   000007 CF xxxx            STX   second
  17   17                             swap  first, second
  18    4m  00000A C6 xxxx    +         LDA   first
  19    5m  00000D CE xxxx    +         LDX   second
  20    6m  000010 C7 xxxx    +         STA   second
  21    7m  000013 CF xxxx    +         STX   first
  22   18   000016 9D                 NOP
  23   19   000017 20EB               BRA   main
  24   20
  25   21                     datSec: SECTION
  26   22   000000                    first:  DS.B  1
  27   23   000001                    second: DS.B  1
```

For the same code, with `MLIST OFF`, the listing file is as shown in Listing 11.44.

Listing 11.44 Assembler output listing of the example in Listing 11.42 with MLIST OFF

```
 Abs. Rel. Loc     Obj. code Source line
 ---- ---- ------ ---------- -----------
   1    1                             XDEF   entry
   3    3                     swap:   MACRO
   4    4                             LDA    \1
   5    5                             LDX    \2
   6    6                             STA    \2
   7    7                             STX    \1
```

```
   8    8                            ENDM
   9    9                      codSec: SECTION
  10   10                      entry:
  11   11   000000 A6F0                LDA #$F0
  12   12   000002 AE0F                LDX #$0F
  13   13                      main:
  14   14   000004 C7 xxxx             STA first
  15   15   000007 CF xxxx             STX second
  16   16                              swap first, second
  21   17   000016 9D                  NOP
  22   18   000017 20EB                BRA main
  23   19                      datSec: SECTION
  24   20   000000             first:  DS.B 1
  25   21   000001             second: DS.B 1
```

The `MLIST` directive does not appear in the listing file. When a macro is called after a `MLIST ON`, it is expanded in the listing file. If the `MLIST OFF` is encountered before the macro call, the macro is not expanded in the listing file.

NOLIST - Disable Listing

Syntax

```
NOLIST
```

Synonym

```
NOL
```

Description

Suppresses the printing of the following instructions in the assembly listing and debug file until a LIST - Enable Listing assembler directive is reached.

Example

See Listing 11.45 for an example of using `LIST` and `NOLIST`.

Listing 11.45 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 11.46.

Listing 11.46 Assembler output listing from the assembler source code in Listing 11.45

```
Assembler
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1    000000 9D          aaa:    NOP
   2    2
   4    4    000001 9D          bbb:    NOP
   5    5    000002 9D                  NOP
   6    6
  12   12    000005 9D          ddd:    NOP
  13   13    000006 9D                  NOP
```

See Also

LIST - Enable Listing assembler directive

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.

Example

Listing 11.47 shows how the `OFFSET` directive can be used to access an element of a structure.

Listing 11.47 Example assembly source code

```
    6    6                                OFFSET 0
    7    7   000000             ID:       DS.B   1
    8    8   000001             COUNT:    DS.W   1
    9    9   000003             VALUE:    DS.L   1
   10   10          0000 0007   SIZE:     EQU    *
   11   11
   12   12                      DataSec:  SECTION
   13   13   000000             Struct:   DS.B   SIZE
   14   14
   15   15                      CodeSec:  SECTION
   16   16                      entry:
   17   17   000003 CE xxxx               LDX    #Struct
   18   18   000006 8600                  LDA    #0
   19   19   000008 6A00                  STA    ID, X
   20   20   00000A 6201                  INC    COUNT, X
   21   21   00000C 42                    INCA
   22   22   00000D 6A03                  STA    VALUE, X
```

When a statement affecting the location counter other than `EVEN`, `LONGEVEN`, `ALIGN`, or `DS` is encountered after the `OFFSET` directive, the offset section is ended. The preceding section is activated again, and the location counter is restored to the next available location in this section (Listing 11.48).

Listing 11.48 Example where the location counter is changed

```
   7    7                         ConstSec: SECTION
   8    8   000000 11             cst1:     DC.B  $11
   9    9   000001 13             cst2:     DC.B  $13
  10   10
  11   11                                   OFFSET 0
  12   12   000000                ID:       DS.B   1
  13   13   000001                COUNT:    DS.W   1
  14   14   000003                VALUE:    DS.L   1
  15   15          0000 0007      SIZE:     EQU    *
  16   16
  17   17   000002 22             cst3:     DC.B  $22
```

In the example above, the `cst3` symbol, defined after the `OFFSET` directive, defines a constant byte value. This symbol is appended to the section `ConstSec`, which precedes the `OFFSET` directive.

ORG - Set Location Counter

Syntax

```
ORG <expression>
```

Synonym

None

Description

The `ORG` directive sets the location counter to the value specified by `<expression>`. Subsequent statements are assigned memory locations starting with the new location counter value. The `<expression>` must be absolute and may not contain any forward, undefined, or external references. The `ORG` directive generates an internal section, which is absolute (see the Sections chapter).

Example

See Listing 11.49 for an example where `ORG` sets the location counter.

Listing 11.49 Using ORG to set the location counter

```
        org   $2000
b1:     nop
b2:     rts
```

Viewing Listing 11.50, you can see that the `b1` label is located at address `$2000` and label `b2` is at address `$2001`.

Listing 11.50 Assembler output listing from the source code in Listing 11.49

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----  ------ ---------   -----------
    1    1                             org   $2000
    2    2  a002000 9D          b1:     nop
    3    3  a002001 81          b2:     rts
```

See also

Assembler directives:

- DC - Define Constant
- DCB - Define Constant Block
- DS - Define Space
- SECTION - Declare Relocatable Section

PAGE - Insert Page break

Syntax

`PAGE`

Synonym

None

Description

Insert a page break in the assembly listing.

Example

The portion of code in Listing 11.51 demonstrates the use of a page break in the assembler output listing.

Listing 11.51 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 11.52.

Listing 11.52 Assembler output listing from the source code in Listing 11.51

```
 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 11.53).

Listing 11.53 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 11.56) about how to decode it.

Example

The string data in Listing 11.54 assembles to the following data (Listing 11.55). The 11 characters in the string are represented by 8 bytes.

Listing 11.54 RAD50 Example

```
  XDEF rad50, rad50Len
DataSection    SECTION
rad50:         RAD50 "Hello World"
rad50Len:      EQU (*-rad50)/2
```

Listing 11.55 Assembler output where 11 characters are contained in eight bytes

```
$32D4 $4D58 $922A $4BA0
```

This C code shown in Listing 11.56 takes the data and prints “Hello World”.

Listing 11.56 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 11.57 demonstrates the definition of a section `aaa`, which is split into two blocks, with section `bbb` in between them.

The location counter associated with the label `zz` is 1, because a `NOP` instruction was already defined in this section at label `xx`.

Listing 11.57 Example of the SECTION assembler directive

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1                      aaa:    SECTION 4
    2    2   000000 9D          xx:     NOP
    3    3                      bbb:    SECTION 5
    4    4   000000 9D          yy:     NOP
    5    5   000001 9D                  NOP
    6    6   000002 9D                  NOP
    7    7                      aaa:    SECTION 4
    8    8   000001 9D          zz:     NOP
```

The optional qualifier `SHORT` specifies that the section is a short section, That means than the objects defined there can be accessed using the direct addressing mode.

For RS08, there are two additional section qualifiers: RS08_SHORT and RS08_TINY. When a section is declared as RS08_SHORT (or RS08_TINY) all the objects defined there can be accessed using the short (and respectively tiny) addressing modes.

The example in Listing 11.58 demonstrates the definition and usage of a `SHORT` section, and uses the direct addressing mode to access the symbol data.

Listing 11.58 Using the direct addressing mode

```
    1    1                      dataSec: SECTION SHORT
    2    2   000000             data:    DS.B   1
    3    3
    4    4                      codeSec: SECTION
    5    5
    6    6                      entry:
    7    7   000000 9C                   RSP
    8    8   000001 A600                 LDA   #0
    9    9   000003 B7xx                 STA   data
```

See also

Assembler directives:

- ORG - Set Location Counter
- DC - Define Constant
- DCB - Define Constant Block
- DS - Define Space

SET - Set Symbol Value

Syntax

```
<label>: SET <expression>
```

Synonym

None

Description

Similar to the EQU - Equate symbol value directive, the `SET` directive assigns the value of the `<expression>` in the operand field to the symbol in the `<label>` field. The `<expression>` must resolve as an absolute expression and cannot include a symbol that is undefined or not yet defined. The `<label>` is an assembly time constant. `SET` does not generate any machine code.

The value is temporary; a subsequent `SET` directive can redefine it.

Example

See Listing 11.59 for examples of the SET directive.

Listing 11.59 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)`, L or S or T`

Synonym

`GLOBAL, PUBLIC`

Description

This directive specifies labels defined in the current module that are to be passed to the linker as labels that can be referenced by other modules linked to the current module.

The number of symbols enumerated in an `XDEF` directive is only limited by the memory available at assembly time.

The S and T size designators are only available for RS08, and result in marking the symbol as short or tiny.

Example

See Listing 11.60 for the case where the `XDEF` assembler directive can specify symbols that can be used by other modules.

Listing 11.60 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` or `S` or `T`.

Synonym

`EXTERNAL`

Description

This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 32-bit values is passed to the linker.

The number of symbols enumerated in an `XREF` directive is only limited by the memory available at assembly time.

The S and T size designators are only available for RS08, and result in marking the symbol as short or tiny.

Example

```
XREF OtherGlobal ; Reference "OtherGlobal" defined in
                 ; another module. (See the XDEF
                ; directive example.)
```

XREFB - External Reference for Symbols located on the Direct Page

Syntax

```
XREFB <symbol>[,<symbol>]...
```

Synonym

None

Description

This directive specifies symbols referenced in the current module but defined in another module. Symbols enumerated in a `XREFB` directive, can be accessed using the direct address mode. The list of symbols and corresponding 8-bit values is passed to the linker.

The number of symbols enumerated in a `XREFB` directive is only limited by the memory available at assembly time.

Example

```
XREFB OtherDirect ; Reference "OtherDirect" def in another
                  ; module (See XDEF directive example.)
```

12

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 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 backslash character (\), followed by a digit (0 - 9) or an uppercase letter (A - Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Consider the following macro definition:

```
MyMacro: MACRO
           DC.\0   \1, \2
         ENDM
```

When this macro is used in a program, e.g.:

```
    MyMacro.B $10, $56
```

the Assembler expands it to:

```
    DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator `\1` through `\9` and `\A` through `\Z`, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument itself is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

Macro argument grouping

To pass text including commas as a single macro argument, the Assembler supports a special syntax. This grouping starts with the [? prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, escape brackets, question marks and backward slashes with a backward slash as prefix.

NOTE This escaping only takes place inside of `[? ?]` arguments. A backslash is only removed in this process if it is just before a bracket `([])`, a question mark `(?)`, or a second backslash `(\)`.

Listing 12.1 Example macro definition

```
MyMacro:  MACRO
            DC    \1
          ENDM
MyMacro1: MACRO
            \1
          ENDM
```

Listing 12.2 has some macro calls with rather complicated arguments:

Listing 12.2 Macro calls for Listing 12.1

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro  [?$10, $56?]?]
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines (Listing 12.3):

Listing 12.3 Macro expansion of Listing 12.2

```
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 12.4):

Listing 12.4 Angle bracket syntax

```
      MyMacro <$10, $56>
```

However, this old syntax is ambiguous as < and > are also used as compare operators. For example, the following code (Listing 12.5) does not produce the expected result:

Listing 12.5 Potential problem using the angle-bracket syntax

```
      MyMacro <1 > 2, 2 > 3> ; Wrong!
```

Because of this the old angle brace syntax should be avoided in new code. There is also and option to disable it explicitly.

See also the -CMacBrackets: Square brackets for macro arguments grouping and the -CMacAngBrack: Angle brackets for grouping Macro Arguments assembler options.

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 12.6):

Listing 12.6 Clear macro definition

```
clear:   MACRO
           LDX    #\1
           LDA    #16
\@LOOP:    CLR    0,X
           INCX
           DECA
           BNE    \@LOOP
        ENDM
```

This macro is called in the application (Listing 12.7):

Listing 12.7 Calling the clear macro

```
         clear     temporary
         clear     data
```

The two macro calls of `clear` are expanded in the following manner (Listing 12.8):

Listing 12.8 Macro call expansion

```
             clear temporary
               LDX   #temporary
               LDA    #16
_00001LOOP:    CLR   0,X
               INCX
               DECA
               BNE   _00001LOOP
             clear data
               LDX   #data
               LDA   #16
_00002LOOP:    CLR   0,X
               INCX
               DECA
               BNE   _00002LOOP
```

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 calls are also supported.

A macro call is limited to the length of one line, i.e., 1024 characters.

13

Assembler Listing File

The assembly listing file is the output file of the Assembler that contains information about the generated code. The listing file is generated when the `-L` assembler option is activated. When an error is detected during assembling from the file, no listing file is generated.

The amount of information available depends upon the following assembler options:

- -L: Generate a listing file
- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No Macro expansion in listing file
- -Li: No included file in listing file

The information in the listing file also depends on following assembler directives:

- LIST - Enable Listing
- NOLIST - Disable Listing
- CLIST - List conditional assembly
- MLIST - List macro expansions

The format from the listing file is influenced by the following assembler directives:

- PLEN - Set Page Length
- LLEN - Set Line Length
- TABS - Set Tab Length
- SPC - Insert Blank Lines
- PAGE - Insert Page break
- NOPAGE - Disable Paging
- TITLE - Provide Listing Title.

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

Page header

The page header consists of three lines:

- The first line contains an optional user string defined in the `TITLE` directive.

  The second line contains the name of the Assembler vendor `(Freescale)` as well as the target processor name - HC(S)08.
- The third line contains a copyright notice.

Listing 13.1 Example page header output

```
Demo Application
Freescale HC08-Assembler
(c) COPYRIGHT Freescale 1991-2005
```

Source listing

The printed columns can be configured in various formats with the -Lasmc: Configure listing file assembler option. The default format of the source listing has the five columns as in :

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 13.2 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                                 LDA \1
```

```
12    3i                              STA \2
13    4i                            ENDM
14   10                   CodeSec: SECTION
15   11                   Start:
16   12                            cpChar char1, char2
17    2m  000000 C6 xxxx    +      LDA char1
18    3m  000003 C7 xxxx    +      STA char2
19   13   000006 9D                NOP
20   14   000007 9D                NOP
```

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 13.3.

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 13.3 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                                 LDA   \1
   12   3i                                 STA   \2
   13   4i                                ENDM
   14   10                      CodeSec:  SECTION
   15   11                      Start:
   16   12                               cpChar char1, char2
   17   2m   000000 C6 xxxx    +           LDA char1
   18   3m   000003 C7 xxxx    +           STA char2
   19   13   000006 9D                   NOP
   20   14   000007 9D                   NOP
```

In the previous example, the line number displayed in the Rel. column. represent the line number of the corresponding instruction in the source file.

`1i` on absolute line number `10` denotes that the instruction `cpChar: MACRO` is located in an included file.

`2m` on absolute line number `17` denotes that the instruction `LDA     char1` is generated by a macro expansion.

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 13.4.

Listing 13.4 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                                LDA   \1
  12    3i                                STA   \2
  13    4i                              ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                               cpChar char1, char2
  17    2m  000000 C6  xxxx   +           LDA   char1
  18    3m  000003 C7  xxxx   +           STA   char2
  19   13   000006 9D                   NOP
  20   14   000007 9D                   NOP
```

In the previous example, the hexadecimal number displayed in the column `Loc.` is the offset of each instruction in the section `codeSec`.

There is no location counter specified in front of the instruction `INCLUDE "macro.inc"` because this instruction does not generate code.

The instruction `LDA     char1` is located at offset 0 from the section `codeSec` start address.

The instruction `STA     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 13.5.

Listing 13.5 Example listing file - Obj. code column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------    -----------
   1    1                      ;------------------------------
   2    2                      ; File: test.o
   3    3                      ;------------------------------
   4    4
   5    5                               XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                               INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                                LDA   \1
  12    3i                                STA   \2
  13    4i                              ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                               cpChar char1, char2
  17    2m  000000 C6  xxxx   +          LDA char1
  18    3m  000003 C7  xxxx   +          STA char2
  19   13   000006 9D                   NOP
  20   14   000007 9D                   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 13.6.

Listing 13.6 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                                 LDA   \1
  12   3i                                 STA   \2
  13   4i                               ENDM
  14   10                      CodeSec: SECTION
  15   11                      Start:
  16   12                               cpChar char1, char2
  17   2m   000000 C6  xxxx   +           LDA char1
  18   3m   000003 C7  xxxx   +           STA char2
  19   13   000006 9D                   NOP
  20   14   000007 9D                   NOP
```

14

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
- Parameter passing scheme
- Return Value
- Accessing assembly variables in an ANSI-C source file
- Accessing ANSI-C variables in an assembly source file
- Invoking an assembly function in an ANSI-C source file
- Support for structured types

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 14.1 summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

Table 14.1 HC08 memory models

Option	Memory Model	Local Data	Global Data	Suggested Use
–Ms	SMALL	SP rel	extended	The SMALL memory model is the default. All pointers and functions are assumed to have 16-bit addresses if not explicitly specified. In the SMALL memory model, code and data must be in the 64k address space.
–Mt	TINY	SP rel	direct	In the TINY memory model, all data including stack must fit into the zero page. Data pointers are assumed to have 8-bit addresses if not explicitly specified with the keyword __far. The code address space is still 64k and function pointers are still 16 bits in length.

NOTE The default pointer size for the compiler is also affected by the memory model chosen.

Parameter passing scheme

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

Return Value

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

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 14.1).

Listing 14.1 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 14.2).

Listing 14.2 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 14.3).

Listing 14.3 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 14.4).

Listing 14.4 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 14.5).

This can also be done in a separate file, included in the assembly source file.

Listing 14.5 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 14.6).

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 14.6 Example of data and constant reference

```
           LDA CConst
         ....
           LDA CData
          ....
```

Invoking an assembly function in an ANSI-C source file

An function implemented in an assembly source file (`mixasm.asm` in Listing 14.7) can be invoked in a C source file (Listing 14.9). 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 14.7 Example of an assembly file: mixasm.asm

```
         XREF CData
         XDEF AddVar
         XDEF ASMData

DataSec: SECTION
ASMData: DS.B 1
CodeSec: SECTION
AddVar:
         ADD CData   ; add CData to the parameter in register A
         STA ASMData ; result of the addition in ASMData
         RTS
```

We recommend that you generate a header file for each assembly source file (Listing 14.7). This header file (`mixasm.h` in Listing 14.8) should contain the interface to the assembly module.

Listing 14.8 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, 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 14.9. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format.)

Listing 14.9 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
}
```

CAUTION Be careful, as the Assembler will not make any checks on the number and type of the function parameters.

The application must be correctly linked.

For these `C` and `*.asm` files, a possible linker parameter file is shown in Listing 14.10.

Listing 14.10 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 assembler option is activated, the Macro Assembler also supports the definition and usage of structured types. This allows an easier way to access ANSI-C structured variable in the Macro Assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type. See Structured type definition.
- Define a structured variable. See Variable definition.
- Declare a structured variable. See Variable declaration.
- Access the address of a field inside of a structured variable. See Accessing a field address
- Access the offset of a field inside of a structured variable. See Accessing a field offset.

NOTE Some limitations apply in the usage of the structured types in the Macro Assembler. See Structured type: Limitations.

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 14.11:

Listing 14.11 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 14.2 shows how the ANSI-C standard types are converted in the assembler notation:

Table 14.2 Converting ANSI-C standard types to assembler notation

ANSI-C type	Assembler Notation
char	DS - Define Space
short	DS.W
int	DS.W
long	DS.L
enum	DS.W
bitfield	-- not supported --
float	-- not supported --
double	-- not supported --
data pointer	DS.W
function pointer	-- not supported --

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 14.12):

`var: typeName`

where:

- `var` is the name of the variable.
- `typeName` is the type associated with the variable.

Listing 14.12 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 14.13 for an example.

Listing 14.13 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 14.14), 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 14.14 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:
          LDA    myData:field3   ; Loads register A with the content of
                                 ; field field3 from variable myData.
```

NOTE The period cannot be used as separator because in assembly language it is a valid character inside of a symbol name.

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 14.15 for an example of using this notation for accessing an offset.

Listing 14.15 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
           LDA myType->field3,X ; Adds the offset of field 'field3'
                                ; (4) to X and loads A with the
                                ; content of the pointed 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.

15

Make Applications

This chapters has the following sections:

- Assembly applications
- Memory maps and segmentation

Assembly applications

This section covers:

- Directly generating an absolute file
- Mixed C and assembly applications

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 15.1).

Listing 15.1 Example assembly source code

```
SECTIONS
  ROM1       = READ_ONLY  0x0200 TO 0x0FFF;
  SpecialROM = READ_ONLY  0x8000 TO 0x8FFF;
  RAM        = READ_WRITE 0x4000 TO 0x4FFF;
PLACEMENT
  DEFAULT_ROM   INTO ROM1;
  mySection     INTO SpecialROM;
  DEFAULT_RAM   INTO RAM;
END
```

16

How to...

This chapter covers the following topics:

- Working with absolute sections
- Working with relocatable sections
- Initializing the Vector table
- Splitting an application into modules
- Using the direct addressing mode to access symbols

Working with absolute sections

An absolute section is a section whose start address is known at assembly time.

(See modules `fiboorg.asm` and `fiboorg.prm` in the demo directory.)

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 16.1).

Listing 16.1 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 label `cst1` is located at address `$A00`, and label `cst2` is located at address `$A01`.

Listing 16.2 Assembler output listing for Listing 16.1

```
1     1                        ORG    $800
2     2  a000800        var:   DS.B   1
3     3                        ORG    $A00
4     4  a000A00 A6     cst1:  DC.B   $A6
5     5  a000A01 BC     cst2:  DC.B   $BC
```

Locate program assembly source code in a separate absolute section (Listing 16.3).

Listing 16.3 Defining an absolute section containing code

```
        XDEF   entry
        ORG    $C00 ; Absolute code section.
entry:
        LDA    cst1 ; Load value in cst1
        ADD    cst2 ; Add value in cst2
        STA    var  ; Store in var
        BRA    entry
```

In the portion of assembly code above, the `LDA` instruction is located at address `$C00`, and the `ADD` instruction is at address `$C03`. See Listing 16.4.

Listing 16.4 Assembler output listing for Listing 16.3

```
    8    8                                ORG    $C00 ; Absolute code
    9    9                         entry:
   10   10   a000C00 C6 0A00              LDA    cst1 ; Load value
   11   11   a000C03 CB 0A01              ADD    cst2 ; Add value
   12   12   a000C06 C7 0800              STA    var  ; Store in var
   13   13   a000C09 20F5                 BRA    entry
   14   14
```

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used.
- The `RSP` instruction can be used to initialize the stack pointer to `$FF`.
- Publish the application's entry point using `XDEF`.
- The programmer should ensure that the addresses specified in the source files are valid addresses for the MCU being used.

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 16.5.

Listing 16.5 Minimal linker parameter file

```
LINK test.abs  /* Name of the executable file generated.      */
NAMES
  test.o       /* Name of the object file in the application. */
END
SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file.
*/
  MY_ROM  = READ_ONLY  0x4000 TO 0x4FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file.
*/
  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. */
```

NOTE There should be no overlap between the absolute sections defined in the assembly source file and the memory areas defined in the PRM file.

NOTE As the memory areas (segments) specified in the PRM file are only used to allocate relocatable sections, nothing will be allocated there when the application contains only absolute sections. In that case you can even specify invalid address ranges in the PRM file.

Working with relocatable sections

A relocatable section is a section which start address is determined at linking time.

Defining relocatable sections in a source file

Define a relocatable section using the `SECTION` directive. See Listing 16.6 for an example of defining relocatable sections.

Listing 16.6 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 16.7.

Listing 16.7 Assembler output listing for Listing 16.6

```
  2    2                    constSec: SECTION ; Relocatable
  3    3   000000 A6        cst1:     DC.B    $A6
  4    4   000001 BC        cst2:     DC.B    $BC
  5    5
  6    6                    dataSec:  SECTION ; Relocatable
  7    7   000000           var:      DS.B   1
```

Locate program assembly source code in a separate relocatable section (Listing 16.8).

Listing 16.8 Defining a relocatable section for code

```
         XDEF   entry
codeSec: SECTION     ; Relocatable code section.
entry:
         LDA    cst1  ; Load value in cst1
         ADD    cst2  ; Add value in cst2
         STA    var   ; Store in var
         BRA    entry
```

In the previous portion of code, the `LDA` instruction is located at an offset 0 from the `codeSec` section start address, and `ADD` instruction at an offset 3 from the `codeSec` section start address.

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used
- The `RSP` instruction can be used to initialize the stack pointer to `$FF`.
- Publish the application's entry point using the `XDEF` directive.

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 16.9.

Listing 16.9 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

Initializing 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.

- Initializing the Vector table in the linker PRM file (recommended),
- Initializing the Vector Table in a source file using a relocatable section, or
- Initializing the Vector Table in a source file using an absolute section.

The HC(S)08 allows 128 entries in the vector table starting at memory location `$FF00` extending to memory location `$FFFF`.

The `Reset` vector is located in `$FFFE`, and the SWI interrupt vector is located in `$FFFC`. From `$FFFA` down to `$FF00` are located the `IRQ[0]` interrupt (`$FFFA`), `IRQ[1]` (`$FFFA`),..., `IRQ[125]` (`$FF00`).

In the following examples, the `Reset` vector, the SWI interrupt and the `IRQ[1]` interrupt are initialized. The `IRQ[0]` interrupt is not used.

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 16.10). All these labels must be published, otherwise they cannot be addressed in the linker `PRM` file.

Listing 16.10 Initializing the Vector table from a PRM File

```
          XDEF  IRQ1Func, SWIFunc, ResetFunc
DataSec:  SECTION
Data:     DS.W 5        ; Each interrupt increments an element
                        ; of the table.
CodeSec:  SECTION

; Implementation of the interrupt functions.
IRQ1Func:
          LDA   #0
          BRA   int
SWIFunc:
          LDA   #4
          BRA   int
ResetFunc:
          LDA   #8
```

```
            BRA    entry
int:
            PSHH
            LDHX   #Data   ; Load address of symbol Data in X
; X <- address of the appropriate element in the tab
Ofset:      TSTA
            BEQ    Ofset3
Ofset2:
            AIX    #$1
            DECA
            BNE    Ofset2
Ofset3:
            INC    0, X    ; The table element is incremented
            PULH
            RTI
entry:
            LDHX   #$0E00 ; Init Stack Pointer to $E00-$1=$DFF
            TXS
            CLRX
            CLRH

            CLI           ; Enables interrupts

loop:       BRA    loop
```

NOTE The `IRQ1Func`, `SWIFunc`, and `ResetFunc` functions are published. This is required, because they are referenced in the linker PRM file.

NOTE The HC08 processor automatically pushes the PC, X, A, and CCR registers on the stack when an interrupt occurs. The interrupt functions do not need to save and restore those registers. To maintain compatibility with the M6805 Family, the H register is not stacked. It is the user's responsibility to save and restore it prior to returning.

NOTE All Interrupt functions must be terminated with an `RTI` instruction

The vector table is initialized using the linker `VECTOR ADDRESS` command (Listing 16.11).

Listing 16.11 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;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
  SSTACK             INTO MY_STACK;
END
INIT ResetFunc
VECTOR ADDRESS 0xFFF8 IRQ1Func
VECTOR ADDRESS 0xFFFC SWIFunc
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 0xFFF8 IRQ1Func` specifies that the address of the `IRQ1Func` function should be written at address `0xFFF8`.

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 16.12.

Listing 16.12 Initializing the Vector Table in source code with a relocatable section

```
           XDEF  ResetFunc
           XDEF  IRQ0Int
DataSec:   SECTION
Data:      DS.W  5 ; Each interrupt increments an element of the table.
CodeSec:   SECTION
; Implementation of the interrupt functions.
IRQ1Func:
           LDA   #0
           BRA   int
SWIFunc:
           LDA   #4
           BRA   int
ResetFunc:
           LDA   #8
           BRA   entry
DummyFunc:
           RTI
int:
           PSHH
           LDHX  #Data  ; Load address of symbol Data in X
           ; X <- address of the appropriate element in the tab
Ofset:     TSTA
           BEQ   Ofset3
Ofset2:
           AIX   #$1
           DECA
           BNE   Ofset2
Ofset3:
           INC   0, X   ; The table element is incremented
           PULH
           RTI
entry:
           LDHX  #$0E00 ; Init Stack Pointer to $E00-$1=$DFF
           TXS
           CLRX
           CLRH
           CLI          ; Enables interrupts
loop:      BRA   loop

VectorTable: SECTION
; Definition of the vector table.
IRQ1Int:   DC.W  IRQ1Func
IRQ0Int:   DC.W  DummyFunc
SWIInt:    DC.W  SWIFunc
ResetInt:  DC.W  ResetFunc
```

NOTE Each constant in the `VectorTable` section is defined as a word (a 2-byte constant), because the entries in the vector table are 16 bits wide.

NOTE In the previous example, the constant `IRQ1Int` is initialized with the address of the label `IRQ1Func`. The constant `IRQ0Int` is initialized with the address of the label `Dummy Func` because this interrupt is not in use.

NOTE All the labels specified as initialization value must be defined, published (using `XDEF`) or imported (using `XREF`) before the vector table section. No forward reference is allowed in the `DC` directive.

NOTE The constant `IRQ0Int` is exported so that the section containing the vector table is linked with the application.

The section should now be placed at the expected address. This is performed in the linker parameter file (Listing 16.13).

Listing 16.13 Example linker parameter file

```
LINK test.abs
NAMES
  test.o+
END
ENTRIES
  IRQ0Int
END
SECTIONS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0B00 TO 0x0CFF;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
/* Define the memory range for the vector table */
  Vector   = READ_ONLY  0xFFF8 TO 0xFFFF;
END
PLACEMENT
  DEFAULT_RAM       INTO MY_RAM;
  DEFAULT_ROM       INTO MY_ROM;
  SSTACK            INTO MY_STACK;
/* Place the section 'VectorTable' at the appropriated address. */
  VectorTable       INTO Vector;
END
INIT ResetFunc
```

NOTE The statement `Vector = READ_ONLY 0xFFF8 TO 0xFFFF` defines the memory range for the vector table.

NOTE The statement `VectorTable INTO Vector` specifies that the vector table should be loaded in the read only memory area `Vector`. This means, the constant `IRQ1Int` will be allocated at address `0xFFF8`, the constant `IRQ0Int` will be allocated at address `0xFFFA`, the constant `SWIInt` will be allocated at address `0xFFFC`, and the constant `ResetInt` will be allocated at address `0xFFFE`.

NOTE The '+' after the object file name switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

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 16.14 for an example.

Listing 16.14 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.
IRQ1Func:
         LDA   #0
         BRA   int
SWIFunc:
         LDA   #4
         BRA   int
ResetFunc:
```

```
         LDA   #8
         BRA   entry
DummyFunc:
         RTI
int:
         PSHH
         LDHX  #Data    ; Load address of symbol Data in X
         ; X <- address of the appropriate element in the tab
Ofset:   TSTA
         BEQ   Ofset3
Ofset2:
         AIX   #$1
         DECA
          BNE   Ofset2
Ofset3:
          INC   0, X    ; The table element is incremented
          PULH
          RTI
entry:
          LDHX  #$0E00 ; Init Stack Pointer to $E00-$1=$DFF
          TXS
          CLRX
          CLRH
          CLI           ; Enables interrupts

loop:     BRA   loop

          ORG   $FFF8
; Definition of the vector table in an absolute section
; starting at address $FFF8.
IRQ1Int:  DC.W  IRQ1Func
IRQ0Int:  DC.W  DummyFunc
SWIInt:   DC.W  SWIFunc
ResetInt: DC.W  ResetFunc
```

The section should now be placed at the expected address. This is performed in the linker parameter file (Listing 16.15).

Listing 16.15 Example linker parameter file for Listing 16.14:

```
LINK test.abs
NAMES
  test.o+
END
SECTIONS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
```

```
  MY_RAM    = READ_WRITE 0x0B00 TO 0x0CFF;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
END
PLACEMENT
  DEFAULT_RAM         INTO MY_RAM;
  DEFAULT_ROM         INTO MY_ROM;
  SSTACK              INTO MY_STACK;
END
INIT ResetFunc
```

NOTE The `'+'` after the object file name switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

Splitting an application into modules

Complex application or application involving several programmers can be split into several simple modules. In order to avoid any problem when merging the different modules, the following rules must be followed.

For each assembly source file, one include file must be created containing the definition of the symbols exported from this module. For the symbols referring to code label, a small description of the interface is required.

Example of an Assembly File (Test1.asm)

See Listing 16.16 for an example `Test1.asm` include file.

Listing 16.16 Separating Code into Modules — Test1.asm

```
          XDEF AddSource
          XDEF Source
DataSec: SECTION
Source:   DS.W 1
CodeSec: SECTION
AddSource:
          RSP
          ADD Source
          STA Source
          RTS
```

Corresponding include file (Test1.inc)

See Listing 16.17 for an example `Test1.inc` include file.

Listing 16.17 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 16.18 is another assembly code file module for this project.

Listing 16.18 Separating Code into Modules—Test2.asm

```
        XDEF  entry
        INCLUDE "Test1.inc"

CodeSec: SECTION
entry:   RSP
         LDA   #$7
          JSR   AddSource
         BRA   entry
```

The application's `*.prm` file should list both object files building the application. When a section is present in the different object files, the object file sections are concatenated into a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the `*.prm` file.

Example of a PRM file (Test2.prm)

Listing 16.19 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,
- Using the direct addressing mode to access exported symbols,
- Defining symbols in the direct page,
- Using the force operator, or
- Using SHORT sections.

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 16.20 Using direct addressing to access external symbols

```
      XREF.B ExternalDirLabel
      XREF   ExternalExtLabel
…
      LDA    ExternalDirLabel ; Direct addressing mode is used.
…
      LDA    ExternalExtLabel ; Extended addressing mode is used.
```

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 16.21 Using direct addressing to access exported symbols

```
     XDEF.B DirLabel
     XDEF   ExtLabel
…
     LDA    DirLabel ; Direct addressing mode is used.
…
     LDA    ExtLabel ; Extended addressing mode is used.
```

Defining symbols in the direct page

Symbols that are defined in the predefined `BSCT` section are always accessed using the direct-addressing mode (Listing 16.22).

Listing 16.22 Defining symbols in the direct page

```
…
          BSCT
DirLabel: DS.B   3
dataSec:  SECTION
ExtLabel: DS.B   5
…
codeSec:  SECTION
…
          LDA   DirLabel ; Direct addressing mode is used.
…
          LDA   ExtLabel ; Extended addressing mode is used.
```

Using the force operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode (Listing 16.23).

The supported force operators are:

- < or `.B` to force direct addressing mode
- > or `.W` to force extended addressing mode.

Listing 16.23 Using a force operator

```
…
dataSec: SECTION
label:   DS.B  5
…
codeSec: SECTION
…
         LDA   <label   ; Direct addressing mode is used.
         LDA   label.B ; Direct addressing mode is used.
…
         LDA   >label   ; Extended addressing mode is used.
         LDA   label.W ; Extended addressing mode is used.
```

Using SHORT sections

Symbols that are defined in a section defined with the `SHORT` qualifier are always accessed using the direct addressing mode (Listing 16.24).

Listing 16.24 Using SHORT sections

```
…
shortSec: SECTION SHORT
DirLabel: DS.B  3
dataSec:  SECTION
ExtLabel: DS.B  5
…
codeSec:  SECTION
…
          LDA   DirLabel ; Direct addressing mode is used.
…
          LDA   ExtLabel ; Extended addressing mode is used.
```

II

Appendices

This document has the following appendices:

- Global Configuration File Entries
- Local Configuration File Entries
- MASM Compatibility
- MCUasm Compatibility

18

Global Configuration File Entries

This appendix documents the sections and entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [Installation] Section
- [Options] Section
- [XXX_Assembler] Section
- [Editor] Section

[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 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 HC08 Assembler, the name of this section would be [HC08_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 *Appearance* check box 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 *Options* check box 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=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%l,%c
```

Example

Listing 18.1 shows a typical `mcutools.ini` file.

Listing 18.1 Typical mcutools.ini file layout

```
[Installation]
Path=c:\Freescale
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%l,%c

[Options]
DefaultDir=c:\myprj

[HC08_Assembler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```

19

Local Configuration File Entries

This appendix documents the sections and entries that can appear in the local configuration file. Usually, you name this file `project.ini`, where *project* is a placeholder for the name of your project.

A `project.ini` file can contains these sections:

- [Editor] Section
- [XXX_Assembler] Section
- Example

[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=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%l,%c
```

[XXX_Assembler] Section

This section documents the entries that can appear in an `[XXX_Assembler]` section of a `project.ini` file.

NOTE *XXX* is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC08 Assembler, the name of this section would be `[HC08_Assembler]`.

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, etc.) 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 `UltraEdit-32: "c:\Programs Files\IDM Software Solutions\UltraEdit-32\uedit32.exe %f -g%l,%c"`

Description

Command line content to open a file. For details, see also Editor Setting Dialog Box.

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

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 19.1 shows a typical layout of the configuration file (usually `project.ini`).

Listing 19.1 Example of a project.ini file

```
[Editor]
Editor_Name=IDF
Editor_Exe=c:\Freescale\prog\idf.exe
Editor_Opts=%f -g%l,%c

[HC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.asm -w2
RecentCommandLine1=fibo.asm
CurrentCommandLine=fibo.asm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=c:\Freescale\prog\idf.exe %f -g%l,%c
```

20

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:

- 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 20.1 Example

```
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 20.1 are also supported for operators:

Table 20.1 Operator notation for MASM compatibility

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

Directives

Table 20.2 enumerates the directives that are supported by the Macro Assembler for compatibility with MASM:

Table 20.2 Supported MASM directives

Operator	Notation	Description
RMB	DS	Defines storage for a variable. Argument specifies the byte size.
RMD	DS 2*	Defines storage for a variable. Argument specifies the number of 2-byte blocks.
RMQ	DS 4*	Defines storage for a variable. Argument specifies the number of 4-byte blocks.
ELSEC	ELSE	Alternate of conditional block.
ENDC	ENDIF	End of conditional block.
NOL	NOLIST	Specify that no subsequent instructions must be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (visible from outside)

Table 20.2 Supported MASM directives

Operator	Notation	Description
PUBLIC	XDEF	Make a symbol public (visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.
XREFB	XREF.B	Import reference to an external symbol located on the direct page.
SWITCH		Allows switching to a previously defined section.
ASCT		Creates a predefined section named id ASCT.
BSCT		Creates a predefined section named id BSCT. Variables defined in this section are accessed using the direct addressing mode.
CSCT		Creates a predefined section named id CSCT.
DSCT		Creates a predefined section named id DSCT.
IDSCT		Creates a predefined section named id IDSCT.
IPSCT		Creates a predefined section named id IPSCT.
PSCT		Creates a predefined section named id PSCT.

21

MCUasm Compatibility

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

MCUasm compatibility mode can be activated, specifying the `-MCUasm` option.

This chapter covers the following topics:

- Labels
- SET directive
- Obsolete directives

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 21.1 Example

```
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 21.2 Example

```
label: SET *
```

When MCUasm compatibility mode is not activated, the previous portion of code generates an error message because the `SET` label can only refer to the absolute expressions.

Obsolete directives

Table 21.1 enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.

Table 21.1 Obsolete directives

Operator	Notation	Description
RMB	DS	Define storage for a variable
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (visible from the outside)
PUBLIC	XDEF	Make a symbol public (visible from the outside)
EXTERNAL	XREF	Import reference to an external symbol.

Index

Symbols

$() 137
${} 137
%(ENV) Modifier 162
%" Modifier 162
%' Modifier 162
%E Modifier 162
%e Modifier 162
%f Modifier 162
%N Modifier 162
%n Modifier 162
%p Modifier 162
* 295

A

A2309 - File not found 46
About dialog box 131
.abs 156
ABSENTRY Directive 300
ABSENTRY, using 64
Absolute assembly
    Successful 66
Absolute Assembly checkbox 61
Absolute Assembly Project 60
Absolute Expression 295
Absolute file 156
Absolute Section 242, 247
ABSPATH 126, 142, 156, 157
Adding a GENPATH 49
Addressing Mode 266, 277
    Direct 266, 277
    Extended 266, 277
    Immediate 266, 277
    Indexed with post-increment 267
    Indexed, 16-bit offset 266
    Indexed, 8-bit offset 266
    Indexed, 8-bit offset with post-increment 267
    Indexed, no offset 266, 277
    Inherent 266, 277
    Memory to memory direct to direct 267
    Memory to memory indexed to direct with post-increment 267
    Memory-to-memory direct-to-indexed with post- increment 267
    Memory-to-memory immediate-to-direct 267
    Relative 266, 277
    Short 277
    Stack pointer, 16-bit offset 267
    Stack pointer, 8-bit offset 266
    Tiny 277
ALIGN Directive 301, 305, 319, 330
Align location counter (ALIGN) 301, 305
Angle brackets for grouping macro arguments (-CMacAngBrack) 170
Application entry point directive (ABSENTRY) 300
Application standard occurrence (-View) 212
ASCT Directive 419
.asm 155
ASMOPTIONS 142
Assembler
    Configuration 116
    File menu 116
    Input File 132, 155
    Menu 117
    Menu bar 115
    Messages 128
    Option 127
    Options Setting Dialog 127
    Output Files 156
    Status Bar 115
    Toolbar 115
Assembler Build Properties Panels 66
Assembler Directives 266
Assembler Main Window 113

Assembler menu 117

B

BASE Directive 301, 306
Begin macro definition (MACRO) 303, 331
Binary Constant 283
board 20
Borrow license feature (-LicBorrow) 199
BSCT Directive 419

C

-Ci 169
CLIST Directive 302, 307
-CMacAngBrack 170
-CMacBrackets 171
Code Generation 83
Code generation 161
Code Section 241
CodeWarrior Editor Configuration 122
CodeWarrior groups 26
CodeWarrior project window 22
CodeWarrior with COM 122
Color
    for error messages 217
    for fatal messages 217
    for information messages 218
    for user messages 219
    for warning messages 219
COM 122
COM Editor Configuration 123
Command-Line Editor configuration 121
-Compat 172
-Compat Directive 315, 324
Compatibility modes (-Compat) 172
{Compiler} 137
Complex Relocatable Expression 295
Conditional assembly (ELSE) 303, 313
Conditional assembly (IF) 303, 324
Conditional assembly (IFcc) 303, 325
Configure address size in listing file (-Lasms) 187
Configure listing file (-Lasmc) 185
Configure maximum macro nesting (-MacroNest) 203
Constant
    Binary 283, 417
    Decimal 283, 417
    Floating point 284
    Hexadecimal 283, 417
    Integer 283
    Octal 283, 417
    String 284
Constant Section 241
COPYRIGHT 143
Create absolute symbols (OFFSET) 338
Create err.log error file (-WErrFile) 215
Create error listing file (-WOutFile) 238
Creating an Absolute Assembly Project 60
CSCT Directive 419
CTRL-S to save 127
Current Directory 136, 144
CurrentCommandLine 409
Cut filenames in Microsoft format to 8.3 (-Wmsg8x3) 216

D

-D 175
Data Section 242
.dbg 157
DC Directive 300, 309
DCB Directive 300, 310
DDE Editor configuration 122
Debug File 157, 328
Decimal Constant 283
Declare relocatable section (SECTION) 344
Default Directory 144, 400
default workspace 20
DEFAULTDIR 144, 155
DefaultDir 400
Define constant (DC) 309
Define constant block (DCB) 310
Define constant block directive (DCB) 300
Define constant directive (DC) 300
Define label (-D) 175
Define space (DS) 311
Define space directive (DS) 300
derivative 20
Directive

ABSENTRY 300
ALIGN 301, 305, 319, 330
ASCT 419
BASE 301, 306
BSCT 419
CLIST 302, 307
-Compat 315, 324
CSCT 419
DC 300, 309
DCB 300, 310
DS 300, 311
DSCT 419
ELSE 303, 313
ELSEC 418
END 301, 314
ENDC 418
ENDFOR 301, 315
ENDIF 303, 316
ENDM 303, 332
EQU 299, 318
EVEN 301, 319
EXTERNAL 419, 422
FAIL 301, 320
FOR 301, 323
GLOBAL 418, 422
IDSCT 419
IF 303, 324
IFC 326
IFcc 303, 325
IFDEF 303, 326
IFEQ 303, 326
IFGE 303, 326
IFGT 303, 326
IFLE 303, 326
IFLT 303, 326
IFNC 303, 326
IFNDEF 303, 326
IFNE 303, 326
INCLUDE 301, 327
IPSCT 419
LIST 302, 328
LLEN 302, 329
LONGEVEN 301, 330
MACRO 303, 331
MEXIT 303, 332
MLIST 302, 334
NOL 418, 422
NOLIST 302, 336
NOPAGE 302, 337
OFFSET 299, 338
ORG 299, 339
PAGE 302, 340
PLEN 302, 341
PSCT 419
PUBLIC 419, 422
RAD50 300, 342
RMB 418, 422
RMD 418
RMQ 418
SECTION 299, 344
SET 346
SPC 302, 347
SWITCH 419
TABS 302, 347
TITLE 302, 347
TTL 418, 422
XDEF 300, 348
XREF 282, 300, 349
XREFB 300, 349, 419
Directives 266
Disable listing (NOLIST) 302, 336
Disable paging (NOPAGE) 302, 337
Disable user messages (-WmsgNu) 232
Display notify box (-N) 205
Do not use environment (-NoEnv) 208
Drag and Drop 132
DS Directive 300, 311
DSCT Directive 419

E

Editor 407
Editor Setting dialog box 118
Editor_Exe 403, 408
Editor_Name 403, 407
Editor_Opts 404, 408
EditorCommandLine 414
EditorDDEClientName 414
EditorDDEServiceName 415

EditorDDETopicName 414
EditorType 413
EDOUT file 158
EDOUT file generation 158
ELSE Directive 303, 313
ELSEC Directive 418
Enable listing (LIST) 302, 328
End assembly (END) 301, 314
End conditional assembly (ENDIF) 303, 316
END Directive 301, 314
End macro definition (ENDM) 303, 332
End of FOR block (ENDFOR) 301, 315
ENDC Directive 418
ENDFOR Directive 301, 315
ENDIF Directive 303, 316
ENDM Directive 303, 332
-ENV 179
ENVIRONMENT 145
Environment
    File 135
Environment Configuration dialog box 126
Environment variables 126, 135, 141
    ABSPATH 126, 142, 156, 157
    ASMOPTIONS 142
    COPYRIGHT 143
    DEFAULTDIR 144, 155
    ENVIRONMENT 136, 145
    ERRORFILE 146
    GENPATH 51, 126, 148, 155, 327
    HIENVIRONMENT 145
    INCLUDETIME 149
    LIBPATH 126
    OBJPATH 126, 150, 156
    SRECORD 156
    TEXTPATH 126, 151
    TMP 152
EQU Directive 299, 318
Equate symbol value (EQU) 318
Error File 157
Error Listing 157
ERRORFILE 146
EVEN Directive 301, 319
Explorer 136
Expression 295
    Absolute 295
    Complex Relocatable 295
    Simple Relocatable 295, 296
EXTERNAL Directive 419, 422
External reference for symbols on direct page (XREFB) 300, 349
External Symbol 282
External symbol definition (XDEF) 300, 348
External symbol reference (XREF) 300, 349

F

-F2 179
-F2o 179
-FA2 179
-FA2o 179
FAIL Directive 301, 320
-Fh 179
Fields
    Label 250
File
    Absolute 156
    Debug 157, 328
    EDOUT 158
    Environment 135
    Error 157
    Include 155
    Input 155
    Listing 157, 302, 328
    Object 156
    PRM 56, 243, 245, 246
    Source 155
File Manager 136
File menu 116
File menu options 116
Floating-Point Constant 284
FOR Directive 301, 323
Force long-word alignment (LONGEVEN) 301, 330
Force word alignment (EVEN) 301, 319

G

Generate error message (FAIL) 301, 320
Generate listing file (-L) 182

Generating Absolute Assembly Using Assembler Build Tool 65
Generating Absolute Assembly Using CodeWarrior IDE 64
GENPATH 48, 49, 51, 126, 148, 155, 327
    Adding 49
GENPATH environment variable 51
GLOBAL Directive 418, 422
Global Editor 118
Global Editor Configuration dialog box 119
Graphic User Interface (GUI) 111
Group 400
Groups, CodeWarrior 26

H

-H 181
HCS08 Assembler > Output Page 30
HCS08 Assembler Build Properties Panels 68
HCS08 Project Properties Window 67
Hexadecimal Constant 283
.hidefaults 135, 136
HIENVIRONMENT 145
HIGH 283
Host 161

I

-I 182
IDE 136
IDSCT Directive 419
IF Directive 303, 324
IFC Directive 326
IFcc Directive 303, 325
IFDEF Directive 303, 326
IFEQ Directive 303, 326
IFGE Directive 303, 326
IFGT Directive 303, 326
IFLE Directive 303, 326
IFLT Directive 303, 326
IFNC Directive 303, 326
IFNDEF Directive 303, 326
IFNE Directive 303, 326
.inc 155
INCLUDE Directive 301, 327
Include file path (-I) 182
Include Files 155
Include text from another file (INCLUDE) 301, 327
INCLUDETIME 149
.ini 116
Input 74, 95
Input file 155
Insert blank lines (SPC) 302, 347
Insert page break (PAGE) 302, 340
Instruction set 250
Integer Constant 283
IPSCT Directive 419

L

-L 182
Label field 250
Language 76, 97, 161
-Lasmc 185
-Lasms 187
-Lc 189
-Ld 191
-Le 193
-Li 195
LIBPATH 126
-Lic 197
-LicA 198
-LicBorrow 199
License information (-Lic) 197
License information about all features (-LicA) 198
-LicWait 200
Line continuation 140
Linker main window 59
List conditional assembly (CLIST) 302, 307
LIST Directive 302, 328
List macro expansions (MLIST) 302, 334
Listing File 157, 302, 328
LLEN Directive 302, 329
Local Editor 119
Local editor configuration dialog box 120
LONGEVEN Directive 301, 330
LOW 283
.lst 157

M

MACRO Directive 303, 331
-MacroNest 203
Macros, user defined 266
-MCUasm 204
mcutools.ini 144
Memory model (-M) 202
Menu bar options 115
Message classes 130
Message format
for batch mode (-WmsgFob) 224
for interactive mode (-WmsgFoi) 226
for no file information (-WmsgFonf) 230
for no position information (-WmsgFonp) 222, 224, 226, 227, 229
Message Settings 128
Message Settings dialog box 128, 129
Message Settings options 129
Messages 161
MEXIT Directive 303, 332
Microcontroller Assembler main window 113
Microcontroller Assembler Message Settings dialog box 131
Microcontroller Assembler Option Settings dialog box 41
Microsoft Developer Studio configuration settings 122
MLIST Directive 302, 334
-MMU
Enable Memory Management Unit (MMU) 205
Modifiers 123
-Ms 202, 364
-Mt 202, 364

N

-N 205
New Project Wizard 20
No beep in case of error (-NoBeep) 206
No debug information for ELF/DWARF files (-NoDebugInfo) 207
No included file in listing file (-Li) 195
No information and warning messages (-W2) 214
No information messages (-W1) 213
No Macro call in listing file (-Lc) 189
No macro definition in listing file (-Ld) 191
No macro expansion in listing file (-Le) 193
-NoBeep 206
-NoDebugInfo 207
-NoEnv 208
NOL Directive 418, 422
NOLIST Directive 302, 336
NOPAGE Directive 302, 337
Number of error messages (-WmsgNe) 230, 234
Number of information messages (-WmsgNi) 231
Number of warning messages (-WmsgNw) 231, 232, 233

O

.o 156
Object File 156
Object filename specification (-ObjN) 208
-ObjN 208
OBJPATH 126, 150, 156
Octal Constant 283
OFFSET Directive 299, 338
Operand 266, 277
Operator 284, 418
Addition 285, 294, 298
Arithmetic AND 418
Arithmetic Bit 298
Arithmetic OR 418
Arithmetic XOR 418
Bitwise 287
Bitwise (unary) 288
Bitwise AND 294
Bitwise Exclusive OR 294
Bitwise OR 294
Division 285, 294, 298
Force 293
HIGH 283, 290
HIGH_6_13 291
Logical 289
LOW 283
MAP_ADDR_6 292
Modulo 285, 294, 298

Multiplication 285, 294, 298
Precedence 294
Relational 289, 294
Shift 287, 294, 298
Shift left 418
Shift right 418
Sign 286, 294, 297
Subtraction 285, 294, 298
Option
Code generation 161
Host 161
Language 161
Messages 161
Output 161
Various 161
Option Settings dialog box 127
Option Settings options 128
Options 400, 413
ORG Directive 299, 339
Output 70, 91, 161
Output file format (-F) 179

P

PAGE Directive 302, 340
PATH 150
Path 399
Path environment variables 126
Path list 139
PLEN Directive 302, 341
Print the assembler version (-V) 211
PRM File 243, 245, 246
PRM file 56
Layout 56
-Prod 209
{Project} 137
project.ini 139
Provide listing title (TITLE) 302, 347
PSCT Directive 419
PUBLIC Directive 419, 422

Q

Qualifiers
SHORT 345

R

RAD50 Directive 300, 342
RAD50-encode string constant directive (RAD50) 300
RAD50-encoded string constants (RAD50) 342
RecentCommandLine 409
Relocatable Section 244
Repeat assembly block (FOR) 301, 323
Reserved Symbol 283
Reset vector 64
RGB color
for error messages (-WmsgCE) 217
for fatal messages (-WmsgCF) 217
for information messages (-WmsgCI) 218
for user messages (-WmsgCU) 219
for warning messages (-WmsgCW) 219, 220
RMB Directive 418, 422
RMD Directive 418
RMQ Directive 418
RS08 Project Properties Window 68

S

.s1 156
.s2 156
.s3 157
Save Configuration dialog box 124
SaveAppearance 401
SaveEditor 402
SaveOnExit 401
SaveOptions 402
Section
Absolute 242, 247
Code 241
Constant 241
Data 242
Relocatable 244
SECTION Directive 299, 344
Sections 241
Select File to Assemble dialog box 44, 65
Select File to Link dialog box 58
Set a message
to disable (-WmsgSd) 234
to error (-WmsgSe) 235

to information (-WmsgSi) 236
to warning (-WmsgSw) 237
SET Directive 346
Set environment variable (-ENV) 179
Set line length (LLEN) 302, 329
Set location counter (ORG) 339
Set message file format
for batch mode (-WmsgFb) 216, 221
for interactive mode (-WmsgFi) 216, 223
Set number base (BASE) 301, 306
Set page length (PLEN) 302, 341
Set symbol value (SET) 346
Set tab length (TABS) 302, 347
Short help (-H) 181
SHORT qualifier 345
ShowTipOfDay 412
Simple Relocatable Expression 295, 296
Source File 155
SPC Directive 302, 347
Special Modifiers 162
Specify project file at startup (-Prod) 209
Square brackets for macro arguments grouping (-CMacBrackets) 171
SRECORD 156
Starting assembler 112
Startup
Configuration 139
Status Bar 115
StatusbarEnabled 410
String Constant 284
-Struct 210
Support for structured types (-Struct) 210
Switch case sensitivity on label names off (-Ci) 169
SWITCH Directive 419
Switch MCUasm compatibility ON (-MCUasm) 204
.sx 157
Symbols 281
External 282
Reserved 283
Undefined 282
User Defined 281
{System} 137

T

TABS Directive 302, 347
Terminate macro expansion (MEXIT) 303, 332
TEXTPATH 126, 151
Tip of the Day 37, 112
Tip of the Day dialog box 112
TipFilePos 412
TITLE Directive 302, 347
TMP 152
Toolbar 115
ToolbarEnabled 410
TTL Directive 418, 422

U

Undefined Symbol 282
UNIX 136
User Defined Symbol 281

V

-V 211
Variables
ABSPATH 126
ENVIRONMENT 136
Environment 126, 135
GENPATH 51, 126
LIBPATH 126
OBJPATH 126
TEXTPATH 126
Various 161
-View 212
View menu 117
View menu options 118

W

-W1 213
-W2 214
Wait for floating license availability (-LicWait) 200
-WErrFile 215
WindowFont 411
WindowPos 411
Windows 136
WinEdit 136, 147

-Wmsg8x3 216
-WmsgCE 217
-WmsgCF 217
-WmsgCI 218
-WmsgCU 219
-WmsgCW 219
-WmsgFb 133, 216
-WmsgFbm 221
-WmsgFbv 221
-WmsgFi 133, 216
-WmsgFim 223
-WmsgFiv 223
-WmsgFob 224
-WmsgFoi 226
-WmsgFonf 230
-WmsgFonp 222, 224, 226, 227, 229
-WmsgNe 230, 234
-WmsgNi 231
-WmsgNu 232
-WmsgNw 231, 232, 233
-WmsgSd 234
-WmsgSe 235
-WmsgSi 236
-WmsgSw 237
-WOutFile 238
Write to standard output (-WStdout) 239
-WStdout 239

X

XDEF Directive 300, 348
XREF Directive 282, 300, 349
XREFB Directive 300, 349, 419