HC(S)08 / RS08
Assembler Manual

Revised: 29 April 2006 - HC08ASMRM - REV 1

freescale™
semiconductor

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

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

How to Contact Us

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

Table of Contents

I Using the HC(S)08/RS08 Assembler

9 Macros 371

10 Assembler Listing File 377

11 Mixed C and Assembler Applications 383

II Appendices

Using the 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
- Conforms to the Freescale Assembly Language Input Standard

Structure of this document

This section has the following chapters:

- “Working with the Assembler” on page 17: A tutorial using the CodeWarrior Development Studio 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” on page 89: A description of the Macro Assembler’s Graphical User Interface (GUI)
- “Environment” on page 115: A detailed description of the Environment variables used by the Macro Assembler
- “Files” on page 137: A description of the input and output file the Assembles uses or generates.

- “Assembler Options” on page 143: A detailed description of the full set of assembler options
- “Sections” on page 241: A description of the attributes and types of sections
- “Assembler Syntax” on page 249: A detailed description of the input syntax used in assembly input files.
- “Assembler Directives” on page 299: A list of every directive that the Assembler supports
- “Macros” on page 371: A description of how to use macros with the Assembler
- “Assembler Listing File” on page 377: A description of the assembler output files
- “Mixed C and Assembler Applications” on page 383: A description of the important issues to be considered when mixing both assembly and C source files in the same project
- “Make Applications” on page 395: A description of special issues for the linker
- “How to ...” on page 397: Examples of assembly source code, linker PRM, and assembler output listings.

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

- “Global Configuration File Entries” on page 419: Description of the sections and entries that can appear in the global configuration file - `mcutools.ini`
- “Local Configuration File Entries” on page 429: Description of the sections and entries that can appear in the local configuration file - *`project`*`.ini`
- “MASM Compatibility” on page 441: Description of extensions for compatibility with the MASM Assembler
- “MCUasm Compatibility” on page 445: Description of extensions for compatibility with the MCUasm Assembler

1

Working with the Assembler

This chapter is primarily a tutorial for creating and managing HC(S)08/RS08 assembly projects with the CodeWarrior Development Studio. 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.

Programming Overview

In general terms, an embedded systems developer programs small but powerful microprocessors to perform specific tasks. These software programs for controlling the hardware is often referred to as firmware. One such 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 which 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 of their embedded microprocessors.

Assembly language instructions are considered as being at a lower level (closer to the hardware) than the essentially standardized C instructions. Programming in C may require some additional assembly instructions to be generated over and beyond what an experienced developer could do in straight assembly language to accomplish the same result. As a result, assembly language 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 like C use compilers to translate the syntax used by the programmer to the machine-language of the microprocessor, whereas assembly language uses assemblers. It is also possible to mix assembly and C source code in a single project. See the Mixed C and Assembler Applications chapter.

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

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

The typical configuration of an Assembler is its association with a Project directory on page 18 and an External Editor on page 18. CodeWarrior 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 CodeWarrior coordinates for your projects. The tools used most frequently within CodeWarrior are its Editor, Compiler, Assembler, Linker, the Simulator/Debugger, and Processor Expert. Most of these “Build Tools” are located in the *prog* subfolder of the CodeWarrior installation. The others are directly integrated into the CodeWarrior Development Studio

The textual statements and instructions of the assembly-language syntax are written by editors. CodeWarrior 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 CodeWarrior or the standalone Assembler.

Project directory

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

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

External Editor

CodeWarrior 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 Build Tools in CodeWarrior to configure or to select your choice of editors. Please refer to the Editor Setting dialog box section of this manual.

Using CodeWarrior to manage an assembly language project

CodeWarrior 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.

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

Listing 1.1 Fibonacci series

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

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

Let’s now rapidly create a project with CodeWarrior and analyze the assembly source and the Linker’s parameter files to calculate a Fibonacci series for a particular 8-bit microprocessor in the Freescale HC(S)08 family - the *MC68HC908GP32*. Along the way, some CodeWarrior tips demonstrate how CodeWarrior could help manage your projects.

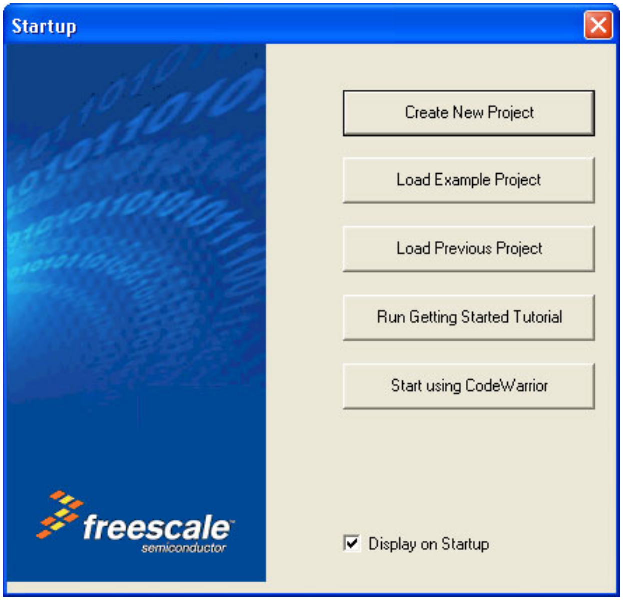
The Wizard

Start the HC(S)08/RS08 Codewarrior IDE application. Its path is:

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

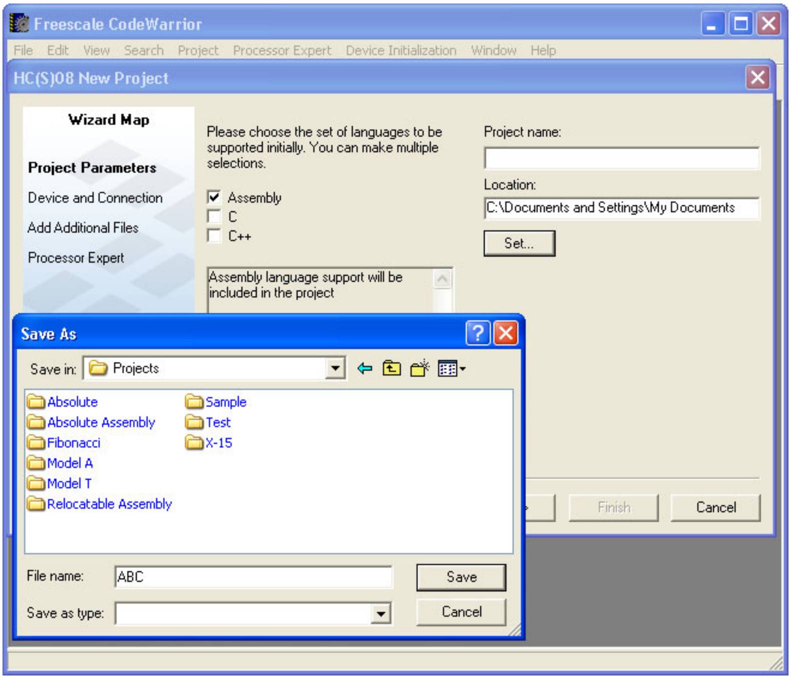
After CodeWarrior opens, press the *Create New Project* button. If CodeWarrior is already running, select *New...* from the *File* menu (*File* > *New...*). See Figure 1.1 on page 19.

Figure 1.1 Startup dialog box



Press the *Create New Project* button. The *HC(S)08 New Project* dialog box appears, showing the *Project Parameters* panel of the *Wizard Map*. If CodeWarrior is already open, select *New...* from the *File* menu (*File* > *New...*). The HC(S) New Project dialog box appears (Figure 1.2 on page 20).

Figure 1.2 HC(S)08 New Project dialog box



Please choose the set of languages to be supported initially. You can make multiple selections.

☑ Assembly
□ C
□ C++

Assembly language support will be included in the project

Project name:

Location: C:\Documents and Settings\My Documents

Save in: Projects

Absolute
Absolute Assembly
Fibonacci
Model A
Model T
Relocatable Assembly
Sample
Test
X-15

File name: ABC

Save as type:

Enter the *Project Parameters* of the *Wizard Map* for your project. For the programming language, check *Assembly* and uncheck both *C* and *C++*. Type the name for the project in the *Project name:* text box. In the event that you want another location for the project directory than the default in the *Location:* text box, press *Set...* and browse to the new location. There is no need to first prepare an empty folder, as CodeWarrior automatically creates its own folder - the *project directory.*

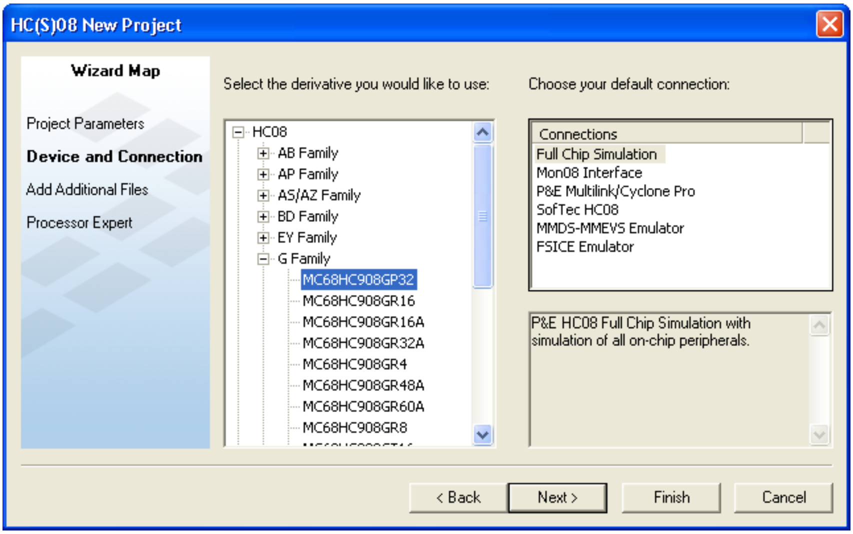
NOTE If you do not use the default *Location* for the project directory, you need not enter a name in the *Project name:* text box. Whatever you enter in the *File name:* text box will be entered into *Location* automatically.

CodeWarrior uses the default `*.mcp` extension, so you do not have to explicitly append any extension to the filename.

Press the *Save* and *Next* > buttons to close the dialog boxes.

The *Device and Connection* dialog box of the *Wizard Map* appears (Figure 1.3 on page 21).

Figure 1.3 Device and Connection dialog box



Select the derivative you would like to use:

- HC08
  - AB Family
  - AP Family
  - AS/AZ Family
  - BD Family
  - EY Family
  - G Family
    - MC68HC908GP32
    - MC68HC908GR16
    - MC68HC908GR16A
    - MC68HC908GR32A
    - MC68HC908GR4
    - MC68HC908GR48A
    - MC68HC908GR60A
    - MC68HC908GR8

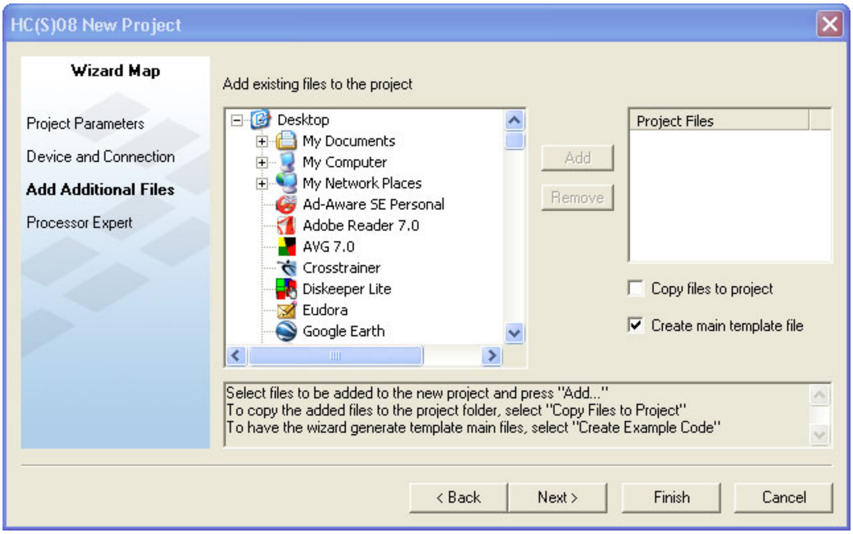
Choose your default connection:

Connections
Full Chip Simulation
Mon08 Interface
P&E Multilink/Cyclone Pro
SofTec HC08
MMDS-MMEVS Emulator
FSICE Emulator

P&E HC08 Full Chip Simulation with simulation of all on-chip peripherals.

Select the desired CPU derivative for the project. Expand *HC08* and *G Family*. In this case, the *MC68HC908GP32* derivative is selected. For *Connections*, select the default - *Full Chip Simulation*. Press *Next >*. The *Add Additional Files* dialog box appears (Figure 1.4 on page 22).

Figure 1.4 Add Additional Files dialog box



□ Copy files to project

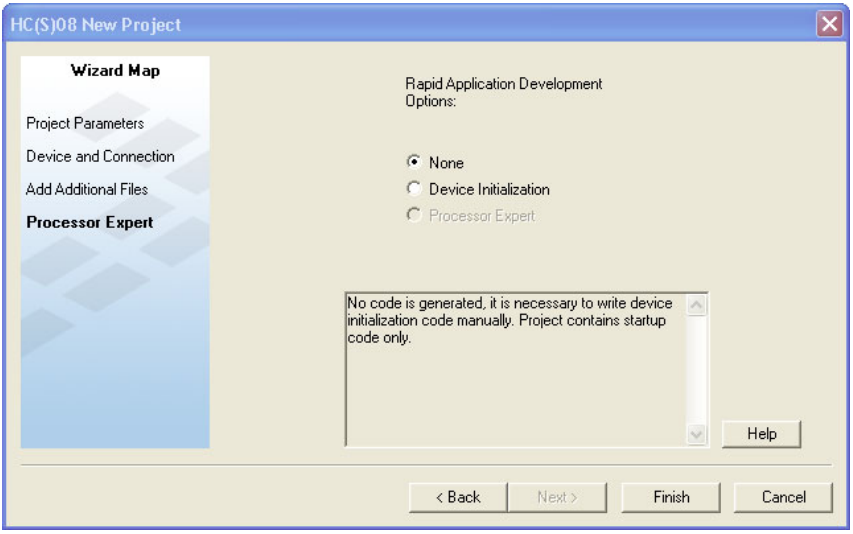
☑ Create main template file

Select files to be added to the new project and press "Add..."
To copy the added files to the project folder, select "Copy Files to Project"
To have the wizard generate template main files, select "Create Example Code"

If you wanted to add any existing files to your project, you could browse in the left panel - *Add existing files to the project* - for the files and press the *Add* button. The added files would then appear in the right panel - *Project Files*. No user files are to be added for this project, so you can either uncheck the *Copy files to project* check box or make sure that no files are selected and leave this check box checked.

Check the *Create main template file* check box. This enables template files including a `main.asm` file in the `Sources` subfolder to be created in the project directory (`ABC`, in this case) with some sample assembly-source code. Press *Next >*. The *Processor Expert* panel appears (Figure 1.5 on page 23).

Figure 1.5 Processor Expert dialog box



Rapid Application Development Options:

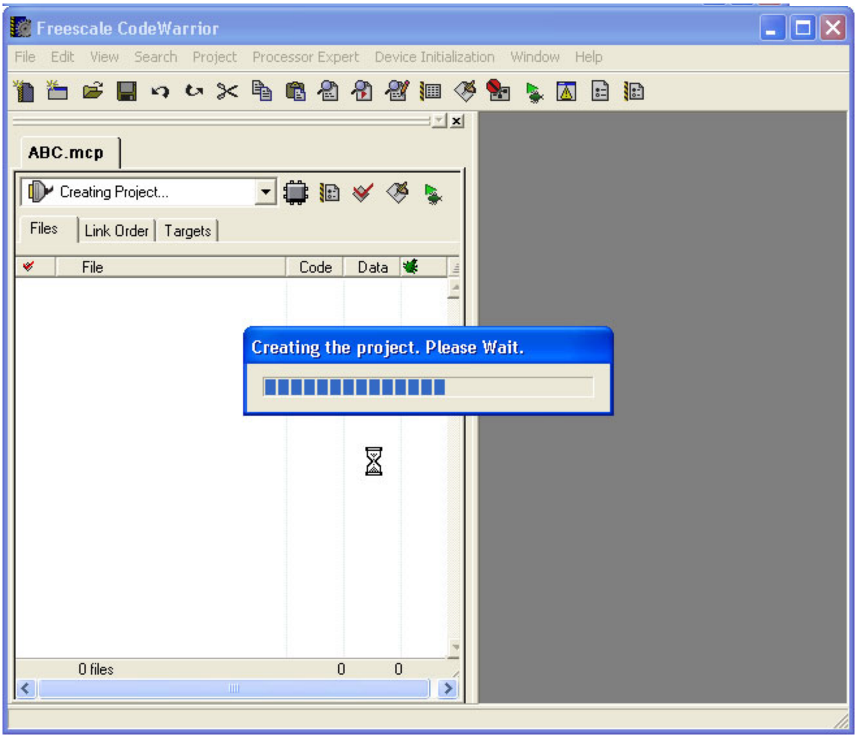
● None
○ Device Initialization
○ Processor Expert

No code is generated, it is necessary to write device initialization code manually. Project contains startup code only.

The default - *None* - is selected. For this simple demo project, you do not need the Rapid Application Development (RAD) tool - *Processor Expert* - in the CodeWarrior Development Studio. A basic demo assembly language project is being created. In practice, you would probably routinely use Processor Expert on account of its many advantages.

Press *Finish* >. The Wizard now creates the project (Figure 1.6 on page 24).

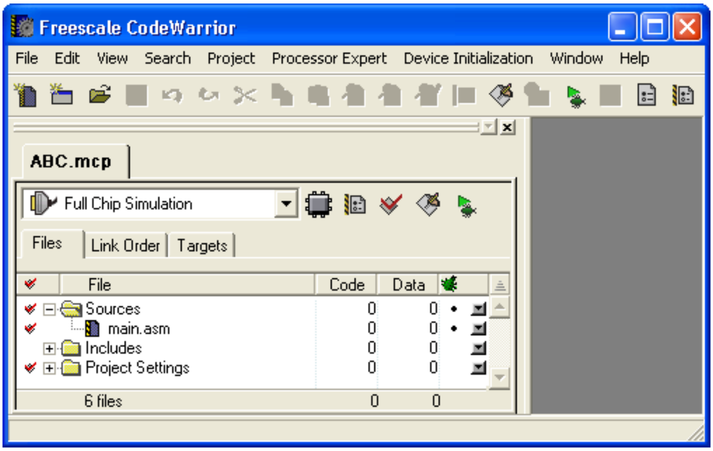
Figure 1.6 The CodeWarrior project is being created...



Creating the project. Please Wait.

Using the Wizard, an HC(S)08 project is set up in a matter of a minute or two. You can add additional components to your project afterwards. A number of files and folders are automatically generated in the root folder that was used in the project-naming process. This folder is referred to in this manual as the project directory. The major GUI component for your project is the project window. The CodeWarrior project window appears (Figure 1.7 on page 25).

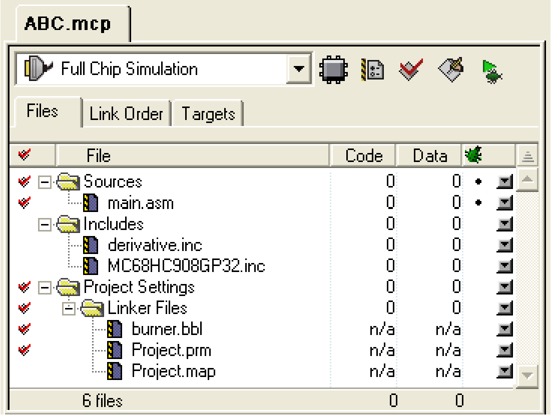
Figure 1.7 CodeWarrior project window


Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Window Help
ABC.mcp
Full Chip Simulation
Files Link Order Targets

✓	File	Code	Data
✓	⊟ Sources	0	0 •
✓	main.asm	0	0 •
	⊞ Includes	0	0
✓	⊞ Project Settings	0	0
	6 files	0	0

If you expand the three “folder” icons, actually groups of files, by clicking in the CodeWarrior project window, you could view some of the files that CodeWarrior created. In general, any files in the project window with red check marks will remain checked until they are successfully assembled, compiled, or linked. At this final stage of the Wizard, you could safely close the project and you can reopen it later. A CodeWarrior project reopens in the same configuration it had when it was last saved (Figure 1.8 on page 25).

Figure 1.8 Project window showing some of the files that the Wizard created



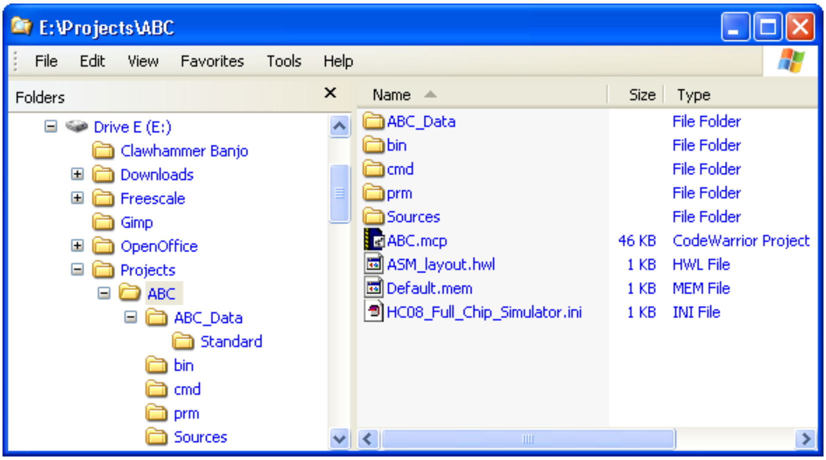
✓	File	Code	Data
✓	⊟ Sources	0	0 •
✓	main.asm	0	0 •
	⊟ Includes	0	0
	derivative.inc	0	0
	MC68HC908GP32.inc	0	0
✓	⊟ Project Settings	0	0
✓	⊟ Linker Files	0	0
✓	burner.bbl	n/a	n/a
✓	Project.prm	n/a	n/a
	Project.map	n/a	n/a
	6 files	0	0

You should examine the types and location of folders and files that CodeWarrior created in the actual project directory so that you know their location if you later configure the Assembler. If you work with standalone tools such as a Compiler, Linker, Simulator/

Debugger, etc., you may need to specify the paths to these files. So it is helpful to know their typical locations and functions.

You could use the Windows Explorer (Figure 1.9 on page 26) to examine the actual folders and files that CodeWarrior created for your project and displays in the project window above. The name and location for the project directory are what you selected when creating the project using the Wizard.

Figure 1.9 Project directory in Windows Explorer



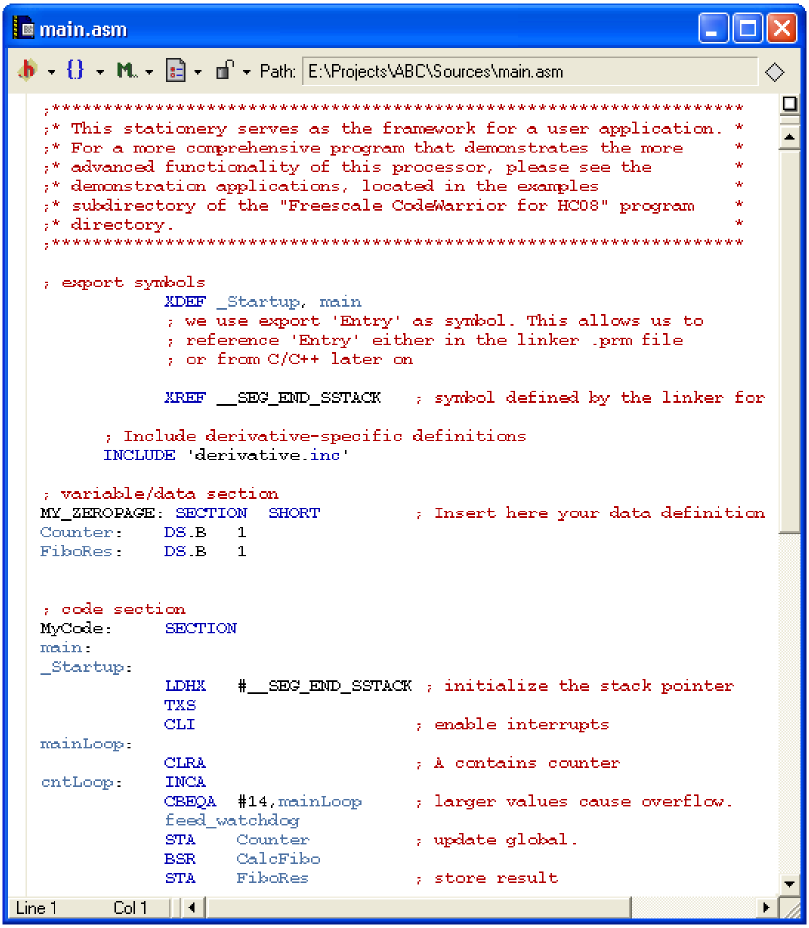
- Drive E (E:)
  - Clawhammer Banjo
  - Downloads
  - Freescale
  - Gimp
  - OpenOffice
  - Projects
    - ABC
      - ABC_Data
        - Standard
      - bin
      - cmd
      - prm
      - Sources

Name	Size	Type
ABC_Data		File Folder
bin		File Folder
cmd		File Folder
prm		File Folder
Sources		File Folder
ABC.mcp	46 KB	CodeWarrior Project
ASM_layout.hwl	1 KB	HWL File
Default.mem	1 KB	MEM File
HC08_Full_Chip_Simulator.ini	1 KB	INI File

The project directory holds a total of six subfolders and 15 files at this point. The major file for any CodeWarrior project is its `<project_name>.mcp` file. This is the file you can use to reopen your project.

Return back to the CodeWarrior project window. Double-click on the `main.asm` file in the *Sources* group. The editor in CodeWarrior opens the `main.asm` file (Figure 1.10 on page 27).

Figure 1.10 Sample main.asm file in the project

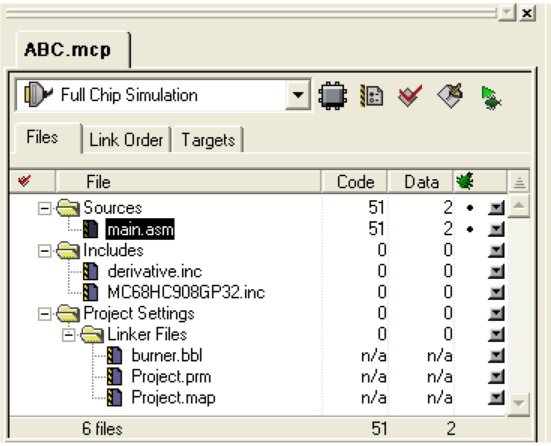


```
;**************************************************************
;* This stationery serves as the framework for a user application. *
;* For a more comprehensive program that demonstrates the more     *
;* advanced functionality of this processor, please see the        *
;* demonstration applications,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

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

            XREF __SEG_END_SSTACK   ; symbol defined by the linker for

      ; 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.
            feed_watchdog
            STA    Counter          ; update global.
            BSR    CalcFibo
            STA    FiboRes          ; store result
```

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.

As a precaution, you can see if the project is configured correctly and if the source code is free of syntactical errors. It is not necessary that you do so, but you should make (build) the default project that CodeWarrior just created. Either press the *Make* button from the toolbar or select (*Project* > *Make*) from the *Project* menu. All of the red check marks will disappear after a successful building of the project (Figure 1.11 on page 28).

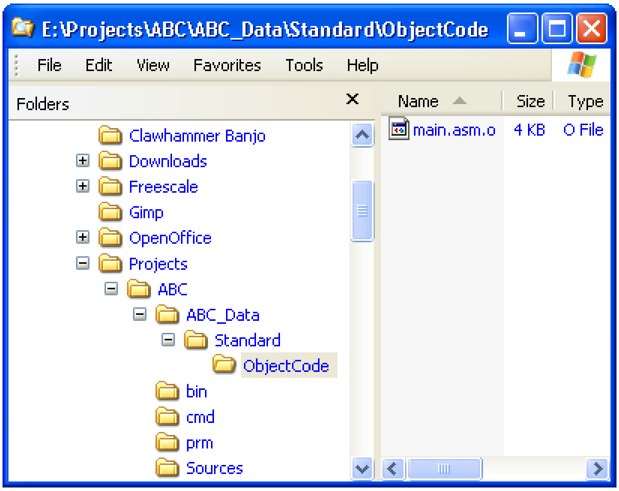
Figure 1.11 Project window after a successful build



File	Code	Data	
Sources	51	2	•
main.asm	51	2	•
Includes	0	0	
derivative.inc	0	0	
MC68HC908GP32.inc	0	0	
Project Settings	0	0	
Linker Files	0	0	
burner.bbl	n/a	n/a	
Project.prm	n/a	n/a	
Project.map	n/a	n/a	
6 files	51	2	

If you checked the project directory after the first successful build (make) of the project with the Windows Explorer, you would see that another subfolder and four additional files were created (Figure 1.12 on page 28).

Figure 1.12 main.o file generated...



- Clawhammer Banjo
- Downloads
- Freescale
- Gimp
- OpenOffice
- Projects
  - ABC
    - ABC_Data
      - Standard
        - ObjectCode
    - bin
    - cmd
    - prm
    - Sources

Name	Size	Type
main.asm.o	4 KB	O File

The new subfolder - *ObjectCode* - holds an object file for every assembly source-code file that is assembled. In this case, the `main.asm.o` object-code file was generated.

Analysis of groups and files in the project window

There are three default groups for this holding this project's files. It really does not matter in which group a file resides as long as that file is somewhere in the project window. A file does not even have to be in any group. The groups do not correspond to any physical folders in the project directory. They are simply present in the project window for conveniently grouping files anyway you choose. You can add, rename, or delete files or groups, or you can move files or groups anywhere in the project window.

CodeWarrior groups

These groups and their usual functions are:

- Sources

  This group contains the assembly source code files.

- Includes

  This group holds include files. One include file is for the particular CPU derivative. In this case, the `MC68HC908GP32.inc` file is for the MC68HC908GP32 derivative.

- Project Settings

  – Linker Files

    This group holds the burner file, the Linker PRM file, and the Linker mapping file.

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 the *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 *bin* subfolder in the project directory when `*.asm` files are assembled with this option set.

TIP To set up your project for generating assembler output listing files, select: *Edit > <target_name> Settings... > Target > Assembler for HC08 > Options > Output*. (The default <*target_name*> is `Standard`.) Check *Generate a listing file*. If you want to format the listing files differently than the default, check *Configure listing file* and make the desired options. You can also add these listing files to the project window for easier viewing instead of having to continually hunt for them. For example, you might add the listing files to the *Sources* group in order to have them near the assembly source files in the project window.

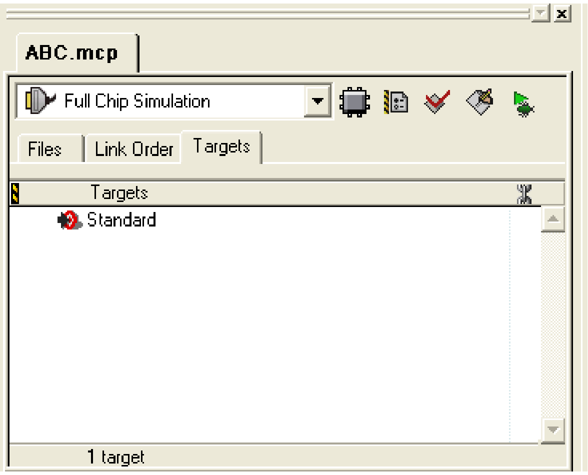
This initial building of your project shows whether it is created and configured correctly. Now you can utilize some of CodeWarrior's features for managing your project. One useful feature is the creation of additional build targets for your projects. You can use multiple targets to have additional subprojects, each with its own files and configuration.

However, it is not at all necessary to use multiple build targets and renaming files and groups in CodeWarrior, so you might skip the following sections about some CodeWarrior project-management features and resume the Assembler part of this tutorial at "Writing your assembly source files" on page 43.

Creating a Target

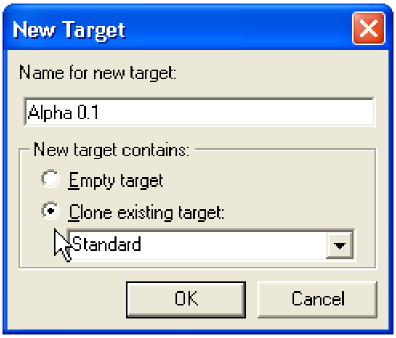
The Wizard created one target which is named `Standard`. You can check this out for yourself by double-clicking on the *Targets* tab in the project window. The *Targets* panel appears (Figure 1.13 on page 30).

Figure 1.13 Targets panel



Creating another build target is easy. Select *Project* > *Create Target....* (If *Create Target...* is grayed in the *Project* menu, click once on the project window and try again.) The *New Target* dialog box appears (Figure 1.14 on page 31).

Figure 1.14 New Target dialog box



New Target

Name for new target:

Alpha 0.1

New target contains:

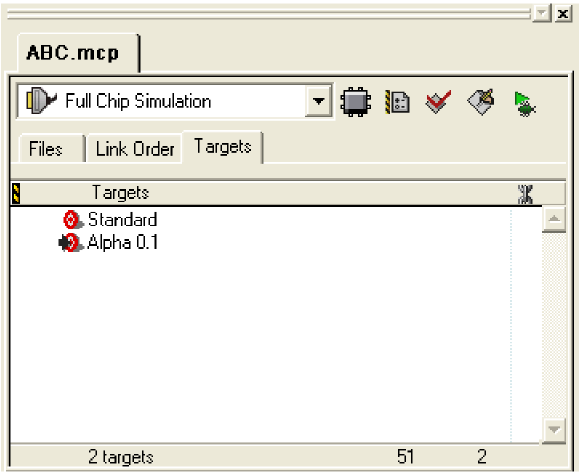
○ Empty target

● Clone existing target:

Standard

Enter the name for the new target and select either of the two options. The *Clone existing target:* option should be used if you plan on using any material from the existing (*Standard*) build target. You can later delete whatever you do not want. Press *OK*. Now there is another build target for your project (Figure 1.15 on page 31).

Figure 1.15 Two build targets are now available



Targets
Standard
Alpha 0.1

2 targets 51 2

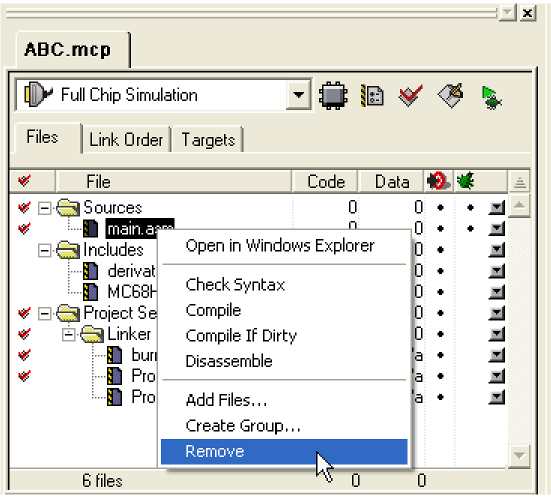
You can use the new target by clicking its icon so that the black arrow is attached to it and then select the *Files* tab. The project window now lists the files used for the new build target. In practice, a number of these files will be the same cloned files used by the other targets, but you can add or delete files as with any build target. You can also select which target is the default upon opening the project by selecting *Project > Set Default Target*.

This project just cloned the default `Standard` build target without changing the configuration. That does not do much at this point but to change the `<target_name>`. So let's create a subfolder in the `Sources` folder and include another `main.asm` file that you can use for your new build target. If you do not create another `main.asm` file in a separate folder, any changes to the original `main.asm` file would affect all build targets.

NOTE In practice, you would rename the files that are not common with files in other build targets to some unique filename for each build target. We will rename them later after you see what might occur when common filenames are used for files that differ among build targets.

One way to have a separate assembler-source file for each project is to remove the original `main.asm` file from the project (both build targets simultaneously) and then add the appropriate `main.asm` file back into each build target. In either build target, right-select the `main.asm` file and select *Remove* from the right-context menu (Figure 1.16 on page 32).

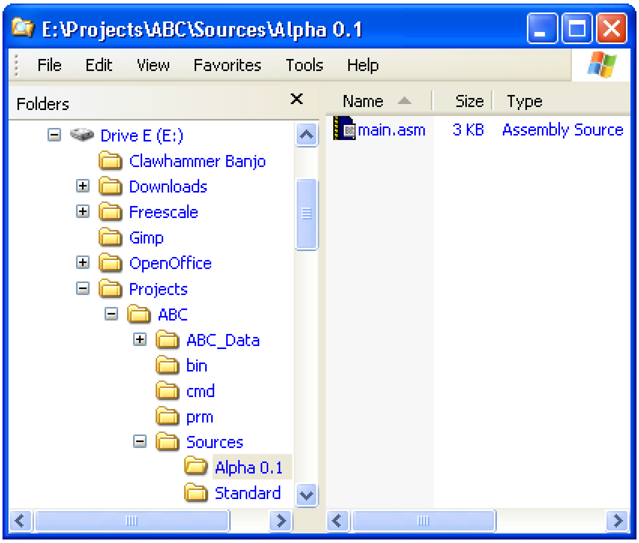
Figure 1.16 Removing the original main.asm file simultaneously from all build targets



A *Freescale CodeWarrior* dialog box appears, asking if you want to remove this file from the project. Press *OK*. The `main.asm` file is now removed from all build targets. However, the `main.asm` still remains in the `Sources` folder in the project directory.

From Windows Explorer, create new subfolders, one for each build target, in the `Sources` folder. You may name them as you choose, but you should use a meaningful name, such as the same name as the appropriate build target. Then cut the `main.asm` file from the `Sources` folder and paste it into every build target's folder (Figure 1.17 on page 33)

Figure 1.17 Project directory with a separate main.asm source file for each build target

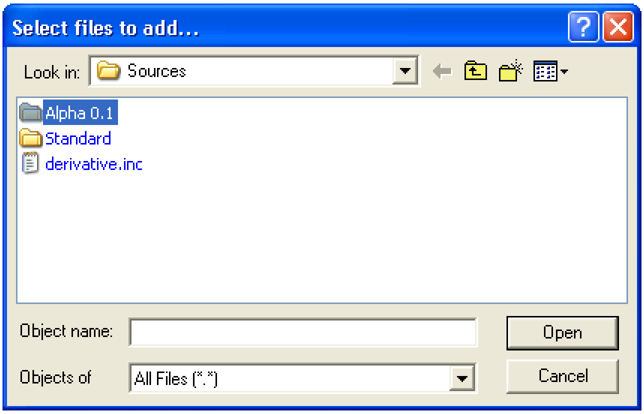


E:\Projects\ABC\Sources\Alpha 0.1

Name	Size	Type
main.asm	3 KB	Assembly Source

Now the appropriate `main.asm` file is added to each build target. In the *Project* menu, select the `Sources` group for any of the build targets and then select *Add Files...*. The *Select files to add...* dialog box appears (Figure 1.18 on page 33).

Figure 1.18 Select files to add... dialog box



Select files to add...

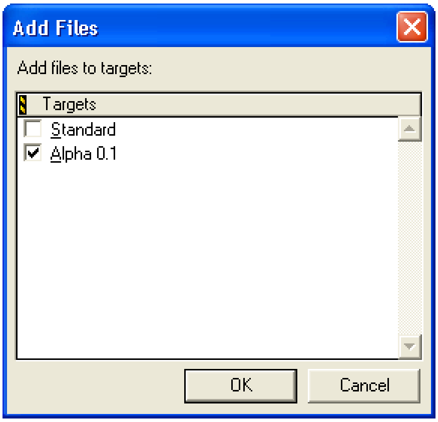
Look in: Sources

Object name:

Objects of: All Files (*.*)

Select the appropriate folder for the build target, press *Open*, and select the `main.asm` file. Press *Open* again. The *Add Files* dialog box appears (Figure 1.19 on page 34).

Figure 1.19 Add Files dialog box

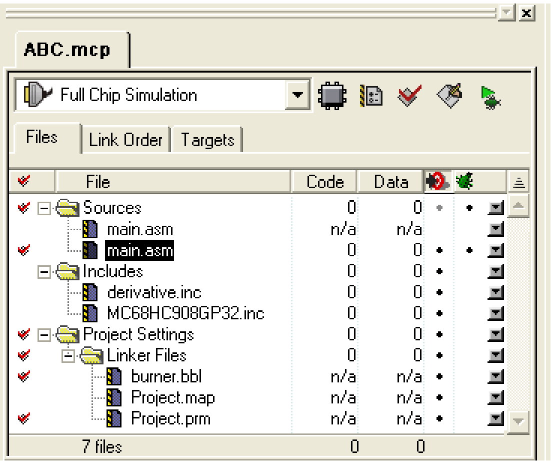


The figure above would be used for the *Alpha 0.1* build target. Deselect the original build target (*Standard*) and keep the new build target (*Alpha 0.1*) checked. Press *OK*. The `main.asm` file is now added to the *Alpha 0.1* build target. Repeat this procedure for adding the `main.asm` to the remaining build target.

Now you can modify a `main.asm` file for one build target without its adversely affecting the other build targets. You could repeat this procedure for any other files in the project that would be different for other build targets. However, you should not do this for those files that are common to all build targets.

NOTE The `main.asm` file was added to each build target, but only one of them is active. The inactive main.asm file will have n/a entries for the *Code* and *Data* columns in the project window (Figure 1.20 on page 35).

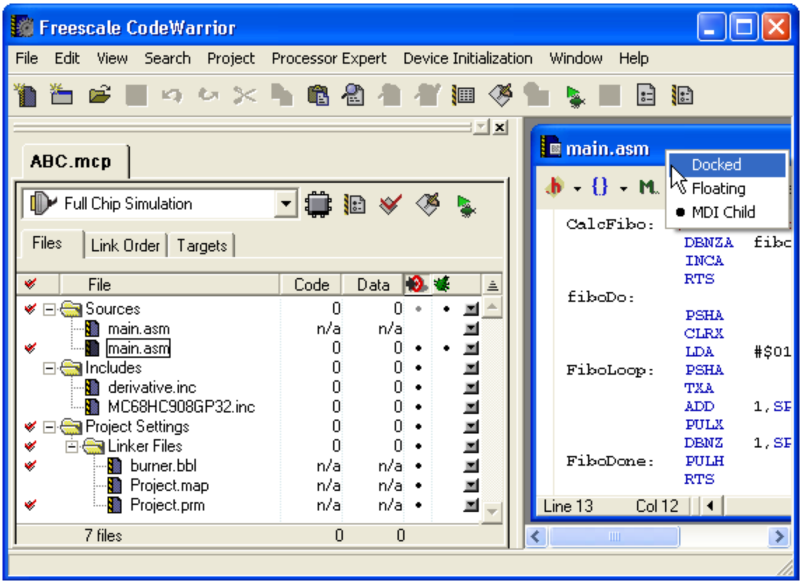
Figure 1.20 Project window showing active and inactive main.asm files



✓	File	Code	Data		
✓	Sources	0	0	•	•
	main.asm	n/a	n/a		
✓	main.asm	0	0	•	•
	Includes	0	0	•	
	derivative.inc	0	0	•	
	MC68HC908GP32.inc	0	0	•	
✓	Project Settings	0	0	•	
✓	Linker Files	0	0	•	
✓	burner.bbl	n/a	n/a	•	
	Project.map	n/a	n/a	•	
✓	Project.prm	n/a	n/a	•	
	7 files	0	0		

So far you have not yet used the editor for this project. For one of the build targets, say the *Alpha 0.1*, double click on the active `main.asm` file in the project window. This file opens. Adjust the mode of the `main.asm` file's window so as to have a comfortable view. One way is to choose the Docked-window option. Right-click on the title bar for the `main.asm` file and select *Docked* in the right-context menu (Figure 1.21 on page 35).

Figure 1.21 Docked-window option for the main.asm file

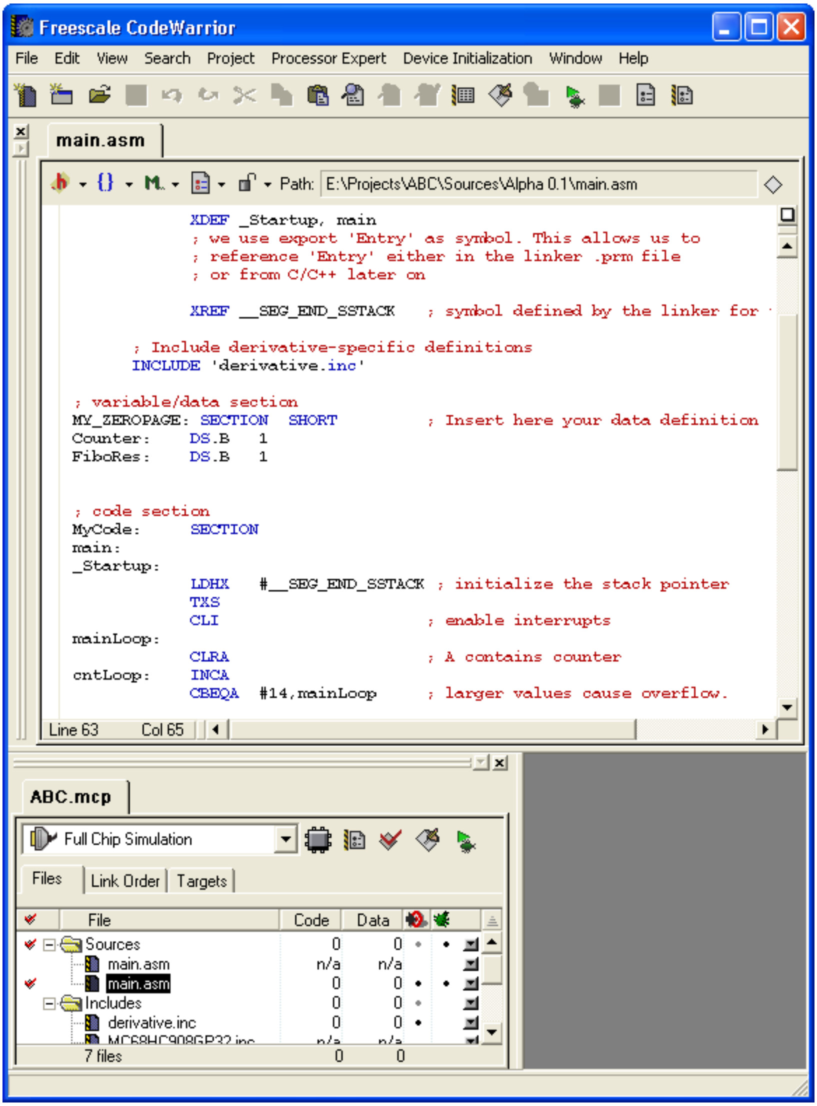


✓	File	Code	Data		
✓	Sources	0	0	•	•
	main.asm	n/a	n/a		
✓	main.asm	0	0	•	•
	Includes	0	0	•	
	derivative.inc	0	0	•	
	MC68HC908GP32.inc	0	0	•	
✓	Project Settings	0	0	•	
✓	Linker Files	0	0	•	
✓	burner.bbl	n/a	n/a	•	
	Project.map	n/a	n/a	•	
✓	Project.prm	n/a	n/a	•	
	7 files	0	0		

```
CalcFibo:
                 DBNZA   fibc
                 INCA
                 RTS
fiboDo:
                 PSHA
                 CLRX
                 LDA     #$01
FiboLoop:        PSHA
                 TXA
                 ADD     1,SP
                 PULX
                 DBNZ    1,SP
FiboDone:        PULH
                 RTS
```

The docked-window view could be adjusted so as to appear as in Figure 1.22 on page 36.

Figure 1.22 Docked-window view for the main.asm file and project window


Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Window Help
main.asm
ABC.mcp
Files Link Order Targets

Path: E:\Projects\ABC\Sources\Alpha 0.1\main.asm

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

            XREF __SEG_END_SSTACK   ; symbol defined by the linker for

      ; 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.
```

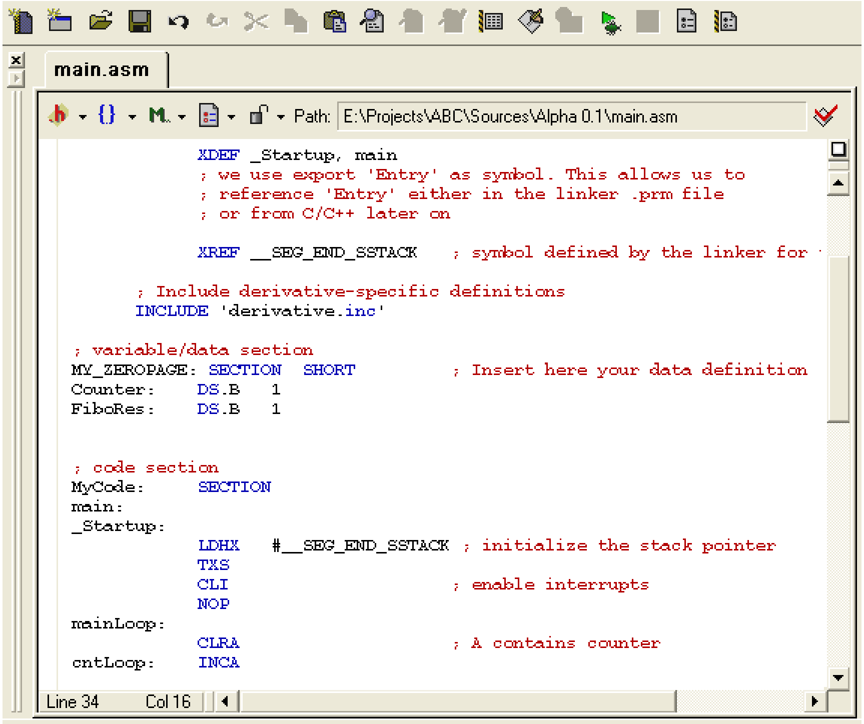
Line 63 Col 65

Full Chip Simulation

✓	File	Code	Data
✓	Sources	0	0
	main.asm	n/a	n/a
✓	main.asm	0	0
	Includes	0	0
	derivative.inc	0	0
	MC68HC908GP32.inc	n/a	n/a
	7 files	0	0

Now you can modify the `main.asm` file in a minor manner. Let’s add a `NOP` instruction after the `CLI` instruction. Place the cursor at the end of the comment in the `CLI` instruction line and press *Enter* on the keyboard. Type `NOP` and press *Enter* once more (Figure 1.23 on page 37).

Figure 1.23 Modified main.asm file



Path: E:\Projects\ABC\Sources\Alpha 0.1\main.asm

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

            XREF __SEG_END_SSTACK   ; symbol defined by the linker for ·

      ; 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
            NOP
mainLoop:
            CLRA                    ; A contains counter
cntLoop:    INCA
```

There are numerous ways to save any changes made by the editor to the `main.asm` file. Some of these are:

- Pressing the *Save* icon on the Toolbar
- Selecting *File > Save* or entering *Ctrl+S* with the keyboard.
- Selecting *Project > Check Syntax (Ctrl+;)*. This also checks the syntax for the `main.asm` file, as the name for the command suggests.
- Selecting *Project > Compile (Ctrl+F7)* or pressing the *Compile* icon on the Toolbar. This also checks the syntax, assembles the `main.asm` file, and produces a `main.asm.o` object-code file in the `bin` folder in the project directory, if successful.
- Selecting *Project > Bring Up To Date (Ctrl+U)*. If successful, this does everything that *Compile* does plus assembling multiple assembly-code files. In addition, each

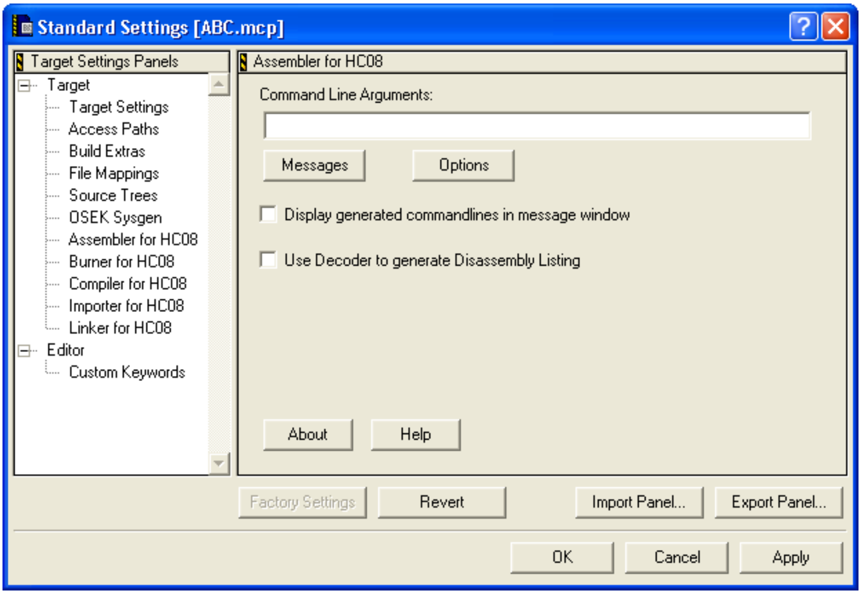
file with a red check mark is processed. However, no executable output (`*.abs`) file is generated.

- Selecting *Project > Make (F7)* or pressing *Make* on any of the two Toolbars. This effects all the functions that *Bring Up To Date* does in addition to generating an executable `*.asm` file in the `bin` folder, if successful
- Selecting *Project > Debug (F5)* or pressing the *Debug* icon on any of the two Toolbars. This does everything that *Make* does in addition to starting the Simulator/ Debugger Build Tool (`hiwave.exe` in the `prog` folder in the CodeWarrior installation folder), if successful.

Generating Listing Files

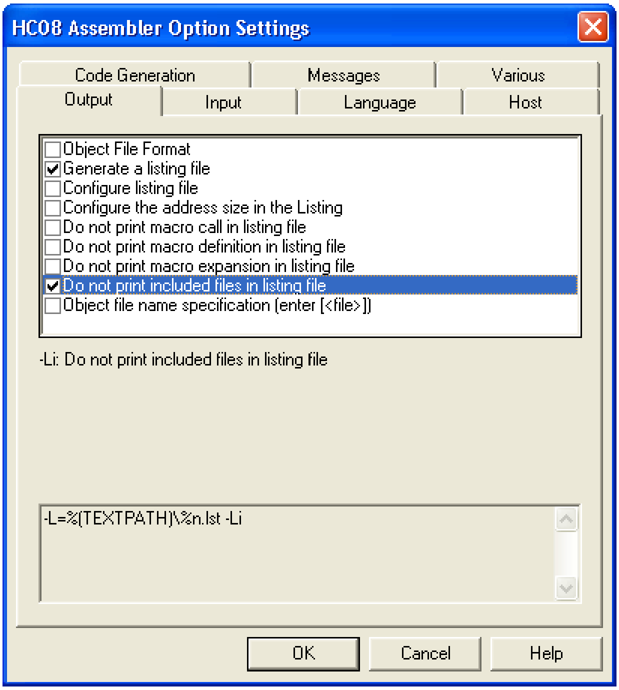
It was mentioned previously that the assembler output listing files were not generated without making configuration changes for the build target. Generating a listing file is easy to set up using Assembler options. Select *Edit > <target_name> Settings > Target > Assembler for HC08*. The *Assembler for HC08* preference panel opens (Figure 1.24 on page 38).

Figure 1.24 Assembler for HC08 preference panel



Press *Options*. The *HC08 Assembler Option Settings* dialog box opens (Figure 1.25 on page 39).

Figure 1.25 HC08 Assembler Option Settings dialog box



□ Object File Format
☑ Generate a listing file
□ Configure listing file
□ Configure the address size in the Listing
□ Do not print macro call in listing file
□ Do not print macro definition in listing file
□ Do not print macro expansion in listing file
☑ Do not print included files in listing file
□ Object file name specification (enter [<file>])

-Li: Do not print included files in listing file

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

Check *Generate a listing file* and also *Do not print included files in lsting file* (unless you actually want to view the sometimes lengthy include files). Press *OK* twice to close the dialog box and the preference panel. Then repeat this procedure for the remaining build targets. With these options set, the Assembler will generate a listing file in the `bin` folder for all `*.asm` files for each build target. The filename for this listing file is the same as the `*.asm` file, but with the `*.lst` file extension.

Using the same filename for the `main.asm` file for all build targets causes a problem for the assembler output listing file. To which `main.asm` file does the `main.lst` listing file correspond? You could eliminate this confusion by choosing a unique filename for the `main.asm` file for each build target. However, using the poor practice of using common filenames for files that differ in other build targets was done intentionally so that:

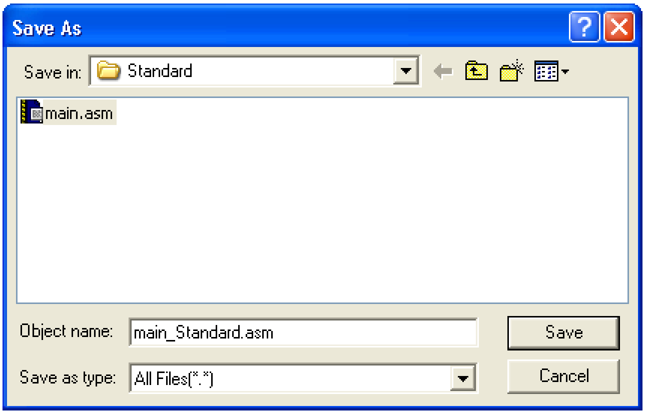
- You could see the confusion it causes with listing files.
- This allows you to employ another CodeWarrior functionality - renaming files.

Renaming files

It is possible to change the name of a file in the project window, add it to the project, and remove the former file from the project window simultaneously.

Double-click on the active `main.asm` file's icon in the project window. The editor opens that file. Select *File > Save as...*. The *Save As* dialog box appears (Figure 1.26 on page 40).

Figure 1.26 Save As dialog box



Save in: Standard

Object name: main_Standard.asm

Save as type: All Files(*.*)

Enter the new filename in the *Object name:* text box. Press *Save*. Close the open file by selecting *File > Close* or by pressing the *Close* button in the Title bar of the open file. Now:

- The new filename (e.g., *`main_Standard.asm`*) replaces the former filename in the project window for all build targets.
- A file with the new filname is created in the folder selected in the *Save As* dialog box <project_name> \<all_source-files>\<build_target>, or in this case: `ABC\Sources\Standard`.

However, the original file still exists in its folder with its original filename.

You can use this procedure for renaming other files in the project window:

- Open the file in the project window that you want to rename.
- Select *File > Save As...*.
- Browse for the folder in which to store the new file.
- Enter a new filename. Press *Save*.

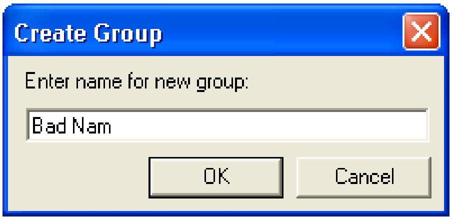
Renaming a filename in this manner simultaneously removes the older file from and imports the newer file into the project (window). Repeat this procedure for the other build targets. You can delete the two unneeded `main.asm` files from the two subfolders, if you choose, as they now longer are involved with the project. You could also delete the `main.lst` listing file and the `main.dbg` file from the `bin` folder if any of them is present.

If you build any of the two build projects from this point, a unique listing file is generated for each build target in the `bin` folder.

Creating a new group

From the *Project* menu, select *Create Group...*. The *Create Group* dialog box appears (Figure 1.27 on page 41).

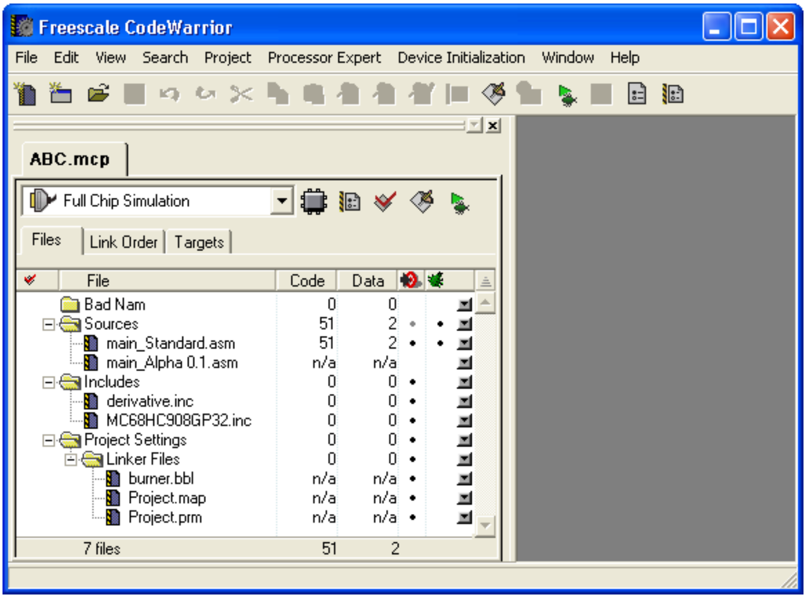
Figure 1.27 Create Group dialog box



Enter name for new group: Bad Nam

Enter a name for the new group in the *Enter name for new group:* text box. Press *OK*. The new group appears in the project window (Figure 1.28 on page 41).

Figure 1.28 Project window now has another group

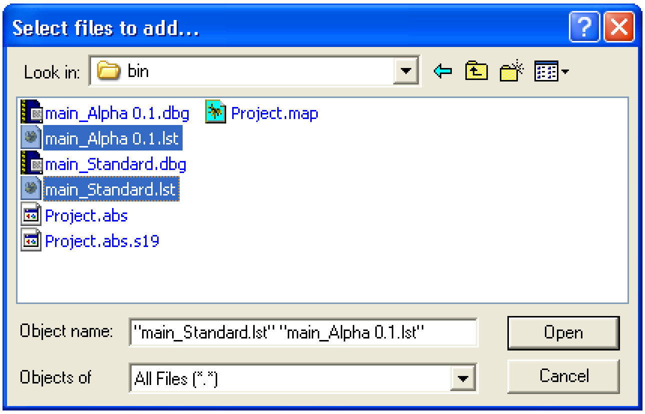


Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Window Help
ABC.mcp
Full Chip Simulation
Files Link Order Targets

✓	File	Code	Data		
	Bad Nam	0	0		
	Sources	51	2	•	•
	main_Standard.asm	51	2	•	•
	main_Alpha 0.1.asm	n/a	n/a		
	Includes	0	0	•	
	derivative.inc	0	0	•	
	MC68HC908GP32.inc	0	0	•	
	Project Settings	0	0	•	
	Linker Files	0	0	•	
	burner.bbl	n/a	n/a	•	
	Project.map	n/a	n/a	•	
	Project.prm	n/a	n/a	•	
	7 files	51	2		

There is only one reason for creating a group: placing one or more files in it. And “Oops!” The group name has an error. But you can rectify that later. Let’s place the two listing files located in the `bin` folder into the new group. (If there are not two listing files - one for each build target, build the build targets until there are two.) Select *Project > Add Files...*.

Figure 1.29 New group - Select files to add... dialog box



Select files to add...

Look in: bin

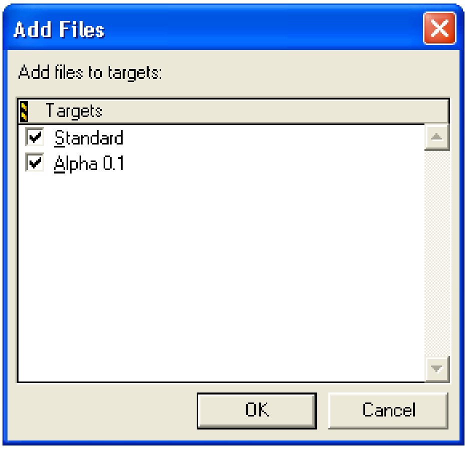
main_Alpha 0.1.dbg
main_Alpha 0.1.lst
main_Standard.dbg
main_Standard.lst
Project.abs
Project.abs.s19
Project.map

Object name: "main_Standard.lst" "main_Alpha 0.1.lst"

Objects of: All Files (*.*)

Select the two listing files and press *Open*. The *Add Files* dialog box appears (Figure 1.30 on page 42).

Figure 1.30 New group - Add Files dialog box



Add Files

Add files to targets:

Targets
☑ Standard
☑ Alpha 0.1

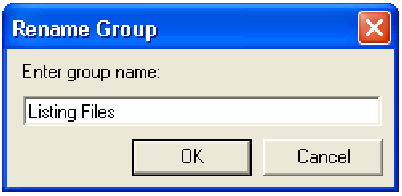
Check all of the build targets - the default. Press *OK*. Now the listing files are conveniently grouped into the new group in the project window.

Renaming groups in the project window

In addition to the ease in changing your *Target Name* or renaming files in the project window, you can also rename any of the groups in the project window.

Double-click on the misnamed group - *Bad Nam*. The *Rename Group* dialog box appears (Figure 1.31 on page 43).

Figure 1.31 Rename Group dialog box

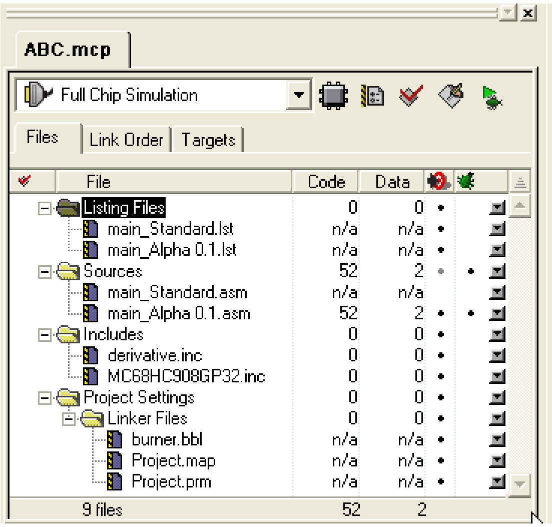


Rename Group

Enter group name: Listing Files

Enter a new name for the group and press *OK*. The group name is now changed in the project window (Figure 1.32 on page 43).

Figure 1.32 Project window with the renamed group



ABC.mcp

Full Chip Simulation

✓	File	Code	Data		
	Listing Files	0	0	•	
	main_Standard.lst	n/a	n/a	•	
	main_Alpha 0.1.lst	n/a	n/a	•	
	Sources	52	2	•	•
	main_Standard.asm	n/a	n/a		
	main_Alpha 0.1.asm	52	2	•	•
	Includes	0	0	•	
	derivative.inc	0	0	•	
	MC68HC908GP32.inc	0	0	•	
	Project Settings	0	0	•	
	Linker Files	0	0	•	
	burner.bbl	n/a	n/a	•	
	Project.map	n/a	n/a	•	
	Project.prm	n/a	n/a	•	
	9 files	52	2		

Writing your assembly source files

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

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

The usual procedure for writing an assembly source-code file is to use the editor that is integrated into CodeWarrior. You can begin a new file by pressing the *New Text File* icon on the Toolbar to open a new file, write your assembly source code, and later save it with a `*.asm` file extension using the *Save* icon on the Toolbar to name and store it wherever you want it placed - usually in the *Sources* folder.

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

Analyzing the project files

We will analyze the default `main.asm` file that was generated when the project was created with the *Wizard*. Listing 1.2 on page 44 is the default but renamed `main_Standard.asm` file that is located in the `Sources` folder created by the *Wizard*. This is the assembler source code for the Fibonacci program.

Listing 1.2 main_Standard.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.                                                      *
;*******************************************************************

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

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

      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.
            feed_watchdog
            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

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

```
;*************************************************************
;*                  Interrupt Vectors                        *
;*************************************************************
            ORG    $FFFA

            DC.W   spurious          ;
            DC.W   spurious          ; SWI
            DC.W   _Startup          ; Reset
```

Since the RS08 memory map is different from the HC08 one (and so is the instruction set), Listing 1.3 on page 46 shows a similar example for RS08.

NOTE In order to assemble files for the RS08 derivative the option -Crs08 should be passed to the assembler. This can be done either directly (in the command line or in the assembler command bar) or by choosing the “Code generation” tab from the assembler options menu. Then select the “Derivative family” option and enable the RS08 Derivative Family radio button.

Listing 1.3 Contents of Example File test_rs08.asm

```
     XDEF Entry ; Make the symbol entry visible for external module
                ; This is necessary to allow the linker to find the
                ; symbol and use it as the entry point for the
                ; application.
cstSec: SECTION ; Define a constant relocatable section
var1: DC.B 5 ; Assign 5 to the symbol var1
dataSec: SECTION ; Define a data relocatable section
data: DS.B 1 ; Define one byte variable in RAM
codeSec: SECTION ; Define a code relocatable section
Entry:
         LDA var1
main:
         INCA
         STA data
         BRA main
```

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

- Make sure that symbols outside of the current source file (in another source file or in the linker configuration file) that are referenced from the current source file are externally visible. Notice that we have inserted the 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 `BSR` or `JSR` instructions in your application. The stack can be initialized in the assembly source code and allocated to RAM memory in the Linker parameter file, if a `*.prm` file is used.

NOTE The default assembly project using the Wizard with CodeWarrior initializes the stack pointer automatically with a symbol defined by the Linker for the end of the stack "`__SEG_END_SSTACK`".

NOTE For the RS08 derivative initializing the stack does not apply.

Assembling your source files

Once an assembly source file is available, you can assemble it. You can either utilize CodeWarrior to assemble the `*.asm` files or alternatively you can use the standalone assembler of the build tools in the prog folder in the CodeWarrior installation.

Assembling with CodeWarrior

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

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

The object files are generated and placed into the `ObjectCode` subfolder in the project directory. The object file (and its path) that results from assembling the `main.asm` file in the default Code Warrior project is:

```
 <project_name>\<project_name>_Data\<build-target_name>\
ObjectCode\main.asm.o.
```

NOTE The build-target name can be changed to whatever you choose in the Target Settings preference panel. Select *Edit* > *<target> Settings...* > *Target* > *Target Settings* and enter the revised target name into the *Target Name:* text box. The default Target Name is *Standard*.

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

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

- `<target_name>.map`

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

- `<target_name>.abs.s19`

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

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

TIP The Wizard does not generate default assembler-output listing files. If you want such listing files generated, you have to select this option: *Edit* > *<target_name> Settings* > *Target* > *Assembler for HC08* > *Options*. Select the *Output* tab in the *HC08 Assembler Option Settings* dialog box. Check *Generate a listing file* and *Do not print included files in list file*. (You can uncheck *Do not print included files in list file* if you choose, but be advised that the include files are usually quite lengthy.) Now a `*.lst` file will be generated in the `bin` subfolder of the project directory whenever a `*.asm` file is assembled.

TIP 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.

Listing 1.4 on page 48 shows the `main.lst` file for this project. The comments are truncated on the far-right edge due to size constraints of the manual’s page.

Listing 1.4 main_Standard.lst assembler output listing file

```
Freescale HC08-Assembler
```

```
(c) Copyright Freescale 1987-2005

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1

;*******************************************************************
    2    2                      ;* This stationery serves as the fram
    3    3                      ;* For a more comprehensive program t
    4    4                      ;* advanced functionality of this pro
    5    5                      ;* demonstration applications, locate
    6    6                      ;* subdirectory of the "Freescale Cod
    7    7                      ;* directory.
    8    8                      ;************************************
    9    9
   10   10                      ; export symbols
   11   11                                  XDEF _Startup, main
   12   12                                  ; we use export 'Entry' a
   13   13                                  ; reference 'Entry' eithe
   14   14                                  ; or from C/C++ later on
   15   15
   16   16                                  XREF __SEG_END_SSTACK   ;
   17   17
   18   18                            ; Include derivative-specific d
   19   19                              INCLUDE 'derivative.inc'
 1238   20
 1239   21                      ; variable/data section
 1240   22                      MY_ZEROPAGE: SECTION  SHORT         ;
 1241   23   000000             Counter:    DS.B   1
 1242   24   000001             FiboRes:    DS.B   1
 1243   25
 1244   26
 1245   27                      ; code section
 1246   28                      MyCode:     SECTION
 1247   29                      main:
 1248   30                      _Startup:
 1249   31   000000 45 xxxx                 LDHX  #__SEG_END_SSTACK
 1250   32   000003 94                      TXS
 1251   33   000004 9A                      CLI                     ;
 1252   34                      mainLoop:
 1253   35   000005 4F                      CLRA                    ;
 1254   36   000006 4C          cntLoop:    INCA
 1255   37   000007 41 0E FB                CBEQA  #14,mainLoop     ;
 1256   38                                  feed_watchdog
 1257   13m  00000A C7 FFFF    +                     STACOPCTL
 1258   39   00000D B7 xx                   STA    Counter          ;
 1259   40   00000F AD 06                   BSR    CalcFibo
 1260   41   000011 B7 xx                   STA    FiboRes          ;
 1261   42   000013 B6 xx                 LDA    Counter
 1262   43   000015 20 EF                 BRA    cntLoop          ;
 1263   44
 1264   45                     CalcFibo:   ; Function to calculate f
 1265   46   000017 4B 02                 DBNZA  fiboDo           ;
 1266   47   000019 4C                    INCA
 1267   48   00001A 81                    RTS
 1268   49                     fiboDo:
 1269   50   00001B 87                    PSHA                    ;
 1270   51   00001C 5F                    CLRX                    ;
 1271   52   00001D A6 01                 LDA    #$01             ;
 1272   53   00001F 87         FiboLoop:   PSHA                    ;
 1273   54   000020 9F                    TXA
 1274   55   000021 9EEB 01               ADD    1,SP
 1275   56   000024 88                    PULX
 1276   57   000025 9E6B 01 F6            DBNZ   1,SP,FiboLoop
 1277   58   000029 8A         FiboDone:   PULH                    ;
 1278   59   00002A 81                    RTS                     ;
 1279   60
 1280   61                     ;*************************************
 1281   62                     ;* spurious - Spurious Interrupt Servi
 1282   63                     ;*             (unwanted interrupt)


Freescale HC08-Assembler
(c) Copyright Freescale 1987-2005

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
 1283   64                     ;************************************
 1284   65                     spurious:                          ;
 1285   66   00002B 9D                    NOP                     ;
 1286   67   00002C 80                    RTI
 1287   68
 1288   69                     ;************************************
 1289   70                     ;*                Interrupt Vectors
 1290   71                     ;************************************
 1291   72                                ORG   $FFFA
 1292   73
 1293   74  a00FFFA xxxx                  DC.W  spurious          ;
 1294   75  a00FFFC xxxx                  DC.W  spurious
 1295   76  a00FFFE xxxx                  DC.W  _Startup          ;
```

Assembling with the Assembler

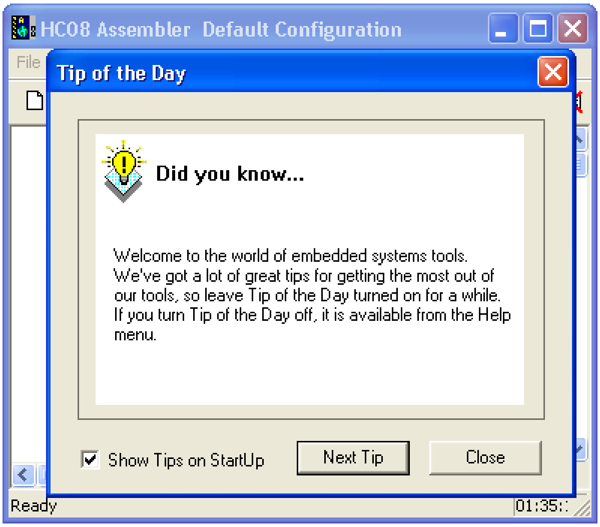
It is also possible to use the HC(S)08 Assembler as a standalone assembler. If you prefer not to use the assembler but do want to use the Linker, you can skip this section and proceed to “Linking the application” on page 67.

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 CodeWarrior Wizard. CodeWarrior can create, configure, and mange 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. You can do this by opening the `ahc08.exe` file in the `prog` folder in the HC08 CodeWarrior installation. The Assembler opens (Figure 1.33 on page 51).

Figure 1.33 HC08 Assembler opens...



HC08 Assembler Default Configuration

Tip of the Day

Did you know...

Welcome to the world of embedded systems tools. We've got a lot of great tips for getting the most out of our tools, so leave Tip of the Day turned on for a while. If you turn Tip of the Day off, it is available from the Help menu.

☑ Show Tips on StartUp

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

Configuring the Assembler

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

- Global

  This data is common to all Build Tools and projects. There may be common data for each Build Tool (Assembler, Compiler, Linker, ...) such as listing the most recent projects, etc. All tools may store some global data into the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (e.g. `C:\WINDOWS`). See Listing 1.5 on page 52.

Listing 1.5 Typical locations for a global configuration file

```
\CW installation directory\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\CW08 V5.x\prog` directory, the initialization file in the same directory as the tool is used:

`C:\Program Files\CW08 V5.x\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.`

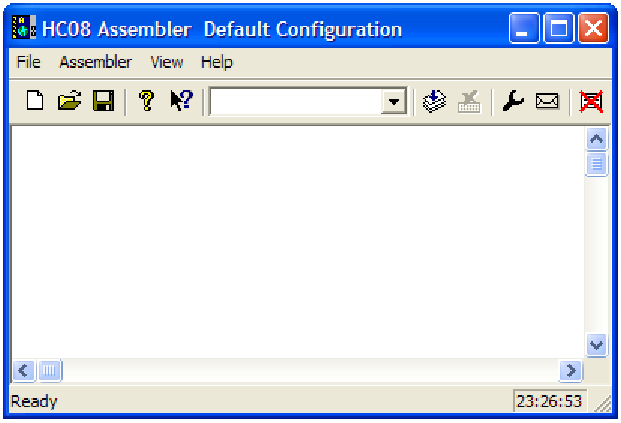
For information about entries for the global configuration file, see Global Configuration File Entries in the Appendices.

- Local

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

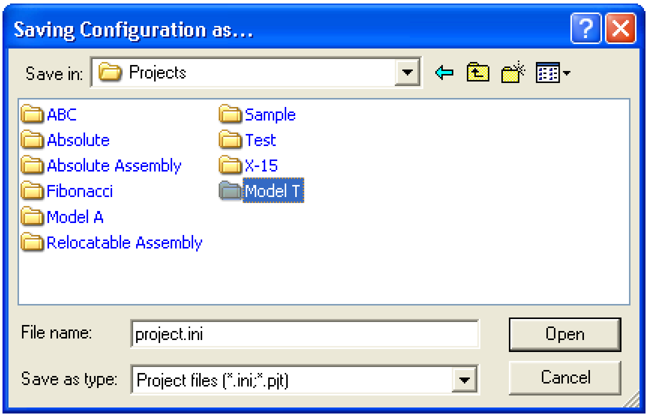
After opening the Assembler, you would load the configuration file for your project if it already had one. However, you will create a new configuration file for the project in this tutorial and save it so that when the project is reopened, its previously saved configuration state will be used. From the *File* menu, select *New / Default Configuration*. The *HC08 Assembler Default Configuration* dialog box appears (Figure 1.34 on page 53)

Figure 1.34 HC08 Assembler New / Default dialog box



Now let’s save this configuration in a newly created folder that will become the project directory. From the *File* menu, select *Save Configuration As...*. A *Saving Configuration as...* dialog box appears. Navigate to the folder of your choice and Click on the *Create New Folder* icon in the Toolbar. Enter a name for the project directory (Figure 1.35 on page 53).

Figure 1.35 Loading configuration dialog box

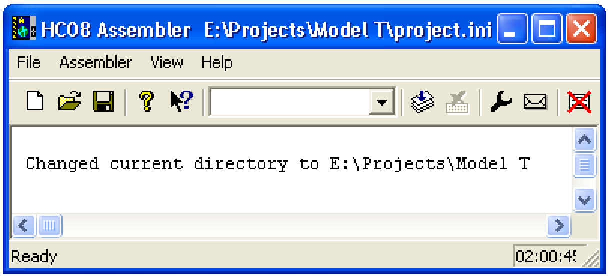


File name: project.ini

Save as type: Project files (*.ini;*.pjt)

Press *Open*. In this case, `Model T` becomes the project directory in the Projects folder. Press *Save* and 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 HC08 Assembler is changed to your project directory (Figure 1.36 on page 54).

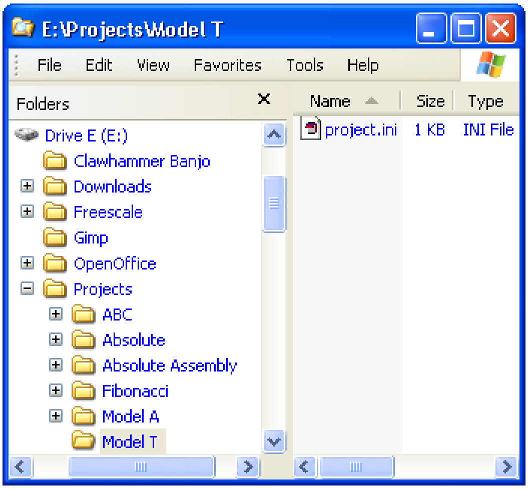
Figure 1.36 Assembler’s current directory switches to your project directory...



```
Changed current directory to E:\Projects\Model T
```

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 1.37 on page 54).

Figure 1.37 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 1.6 on page 54).

Listing 1.6 Contents of the project.ini file

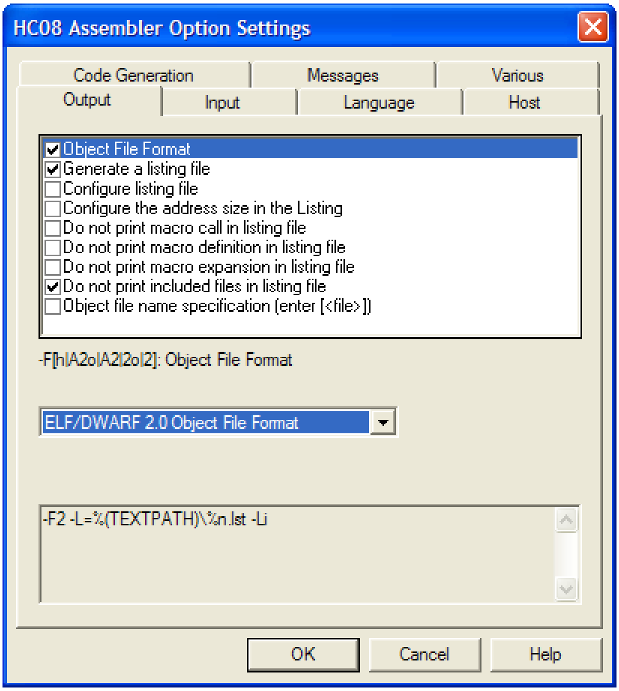
```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,66,87,505,453
```

```
EditorType=4
```

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

Next, you set the object-file format that you will use (HIWARE or ELF/DWARF). Select *Assembler > Options*. The Assembler displays the *HC08 Assembler Option Settings* dialog box (Figure 1.38 on page 55).

Figure 1.38 .HC08 Assembler Option Settings dialog box



☑ Object File Format
☑ Generate a listing file
□ Configure listing file
□ Configure the address size in the Listing
□ Do not print macro call in listing file
□ Do not print macro definition in listing file
□ Do not print macro expansion in listing file
☑ Do not print included files in listing file
□ Object file name specification (enter [<file>])

-F[h|A2o|A2|2o|2]: Object File Format

ELF/DWARF 2.0 Object File Format

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

In the *Output* panel, select the check boxes labeled *Generate a listing file* and *Object File Format*. For the *Object File Format*, select the *ELF/DWARF 2.0 Object File Format* in the pull-down menu. The listing file could be much shorter if the *Do not print included files in list file* check box is checked, so you may want to select that option also. Press *OK* to close the *HC08 Assembler Option Settings* dialog box.

NOTE 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 1.7 on page 56).

Listing 1.7 project.ini file after some assembly options were added

```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,66,87,482,269
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_Standard.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, the `main_Standard.asm` file should be slightly modified so that it appears as below (Listing 1.8 on page 56):

Listing 1.8 main.asm_Standard 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.                                                       *
;********************************************************************

; 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
```

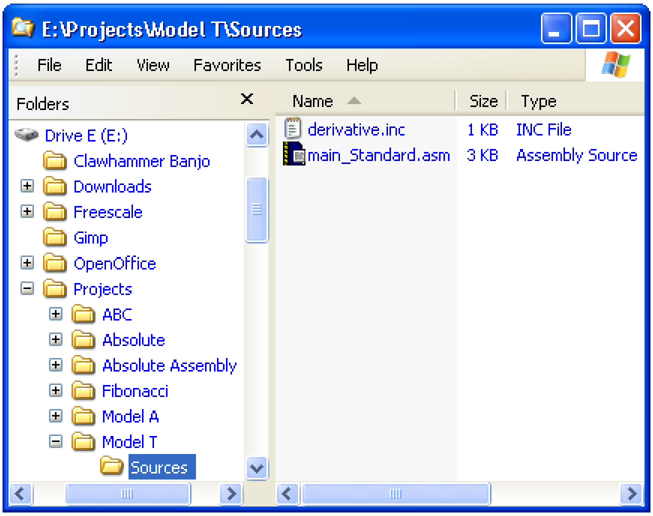
```
      Include derivative-specific definitions

      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 1.39 on page 58):

- the `project.ini` configuration file and

- two files in the `Sources` folder:
  - `main_Standard.asm`
  - `derivative.inc`.

Figure 1.39 Project files

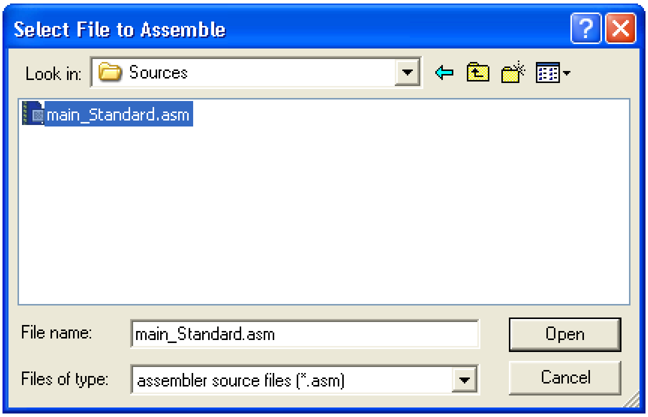


Name	Size	Type
derivative.inc	1 KB	INC File
main_Standard.asm	3 KB	Assembly Source

Assembling the assembly source-code files

Let’s assemble the `main_Standard.asm` file. From the *File* menu, select *Assemble*. The *Select File to Assemble* dialog box appears (Figure 1.40 on page 59).

Figure 1.40 Select Files to Assemble dialog box



Look in: Sources

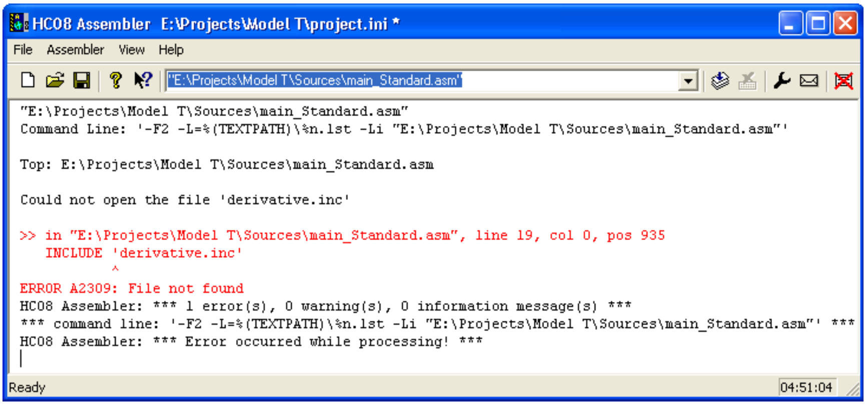
main_Standard.asm

File name: main_Standard.asm

Files of type: assembler source files (*.asm)

Browse to the `Sources` folder in the project directory and select the `main_Standard.asm` file. Press *Open* and the `main.asm` file should start assembling (Figure 1.41 on page 59).

Figure 1.41 Results of assembling the main.asm file...



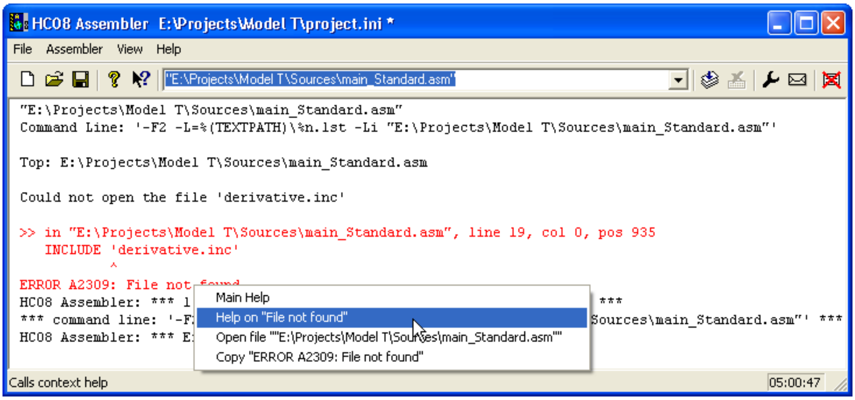
"E:\Projects\Model T\Sources\main_Standard.asm"

```
"E:\Projects\Model T\Sources\main_Standard.asm"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"'

Top: E:\Projects\Model T\Sources\main_Standard.asm

Could not open the file 'derivative.inc'

>> in "E:\Projects\Model T\Sources\main_Standard.asm", line 19, col 0, pos 935
   INCLUDE 'derivative.inc'
           ^
ERROR A2309: File not found
HC08 Assembler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"' ***
HC08 Assembler: *** Error occurred while processing! ***
```

The project window provides information about the assembly process or generates error messages if the assembly was unsuccessful. In this case an error message is generated. - the *A2209 File not found* message. If you right-click on the text containing the error message, a context menu appears (Figure 1.42 on page 60).

NOTE If you get any other types of errors, make sure the `main_Standard.asm` file is modified as shown in Listing 1.8 on page 56

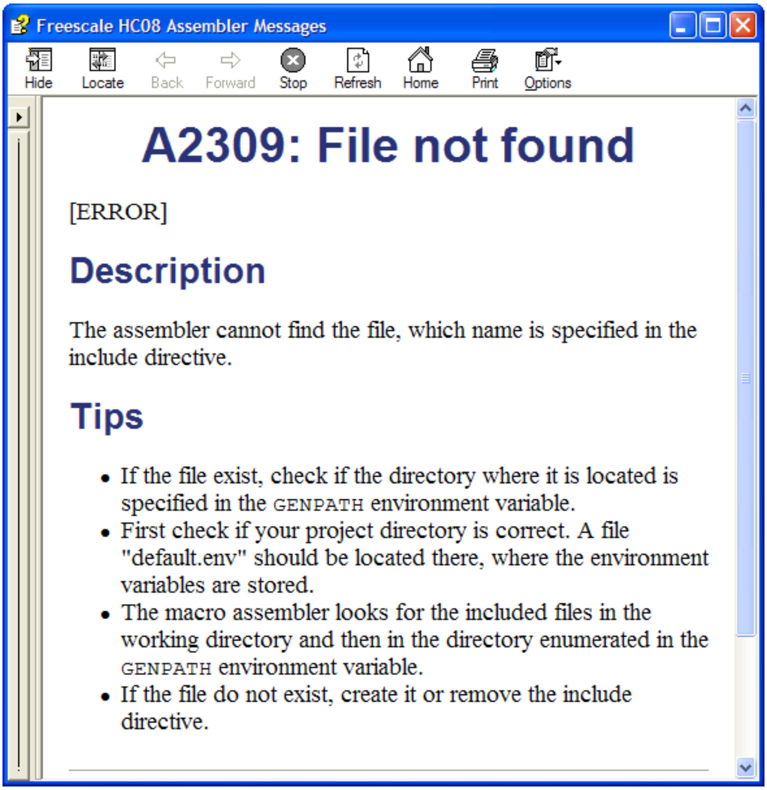
Figure 1.42 Context menu


HC08 Assembler E:\Projects\Model T\project.ini *
File
Assembler
View
Help
"E:\Projects\Model T\Sources\main_Standard.asm"
Main Help
Help on "File not found"
Open file ""E:\Projects\Model T\Sources\main_Standard.asm""
Copy "ERROR A2309: File not found"
Calls context help
05:00:47

```
"E:\Projects\Model T\Sources\main_Standard.asm"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"'

Top: E:\Projects\Model T\Sources\main_Standard.asm

Could not open the file 'derivative.inc'

>> in "E:\Projects\Model T\Sources\main_Standard.asm", line 19, col 0, pos 935
   INCLUDE 'derivative.inc'
           ^
ERROR A2309: File not found
HC08 Assembler: *** 1                                        ***
*** command line: '-F                                        Sources\main_Standard.asm"' ***
HC08 Assembler: *** E
```

Select *Help on "file not found"* and help for the A2309 error message appears (Figure 1.43 on page 61).

Figure 1.43 A2309: File not found



A2309: File not found

[ERROR]

Description

The assembler cannot find the file, which name is specified in the include directive.

Tips

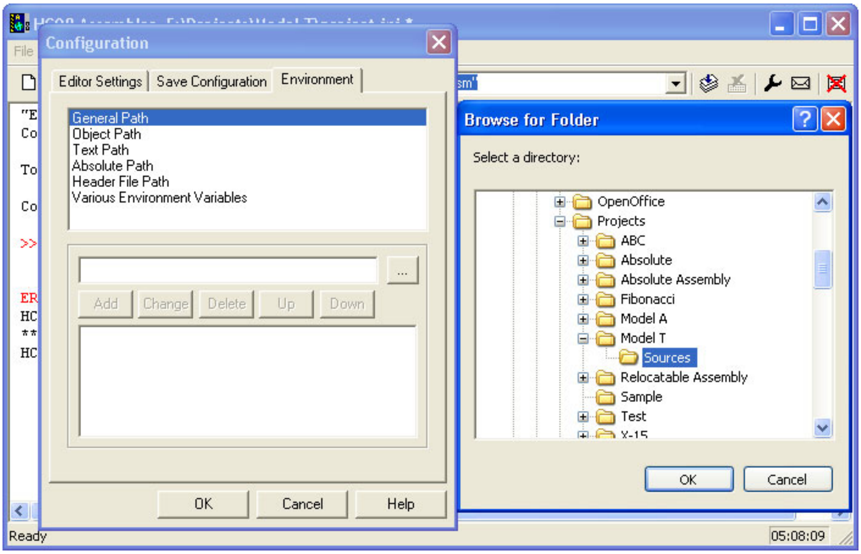
- If the file exist, check if the directory where it is located is specified in the `GENPATH` environment variable.
- First check if your project directory is correct. A file "default.env" should be located there, where the environment variables are stored.
- The macro assembler looks for the included files in the working directory and then in the directory enumerated in the `GENPATH` environment variable.
- If the file do not exist, create it or remove the include directive.

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

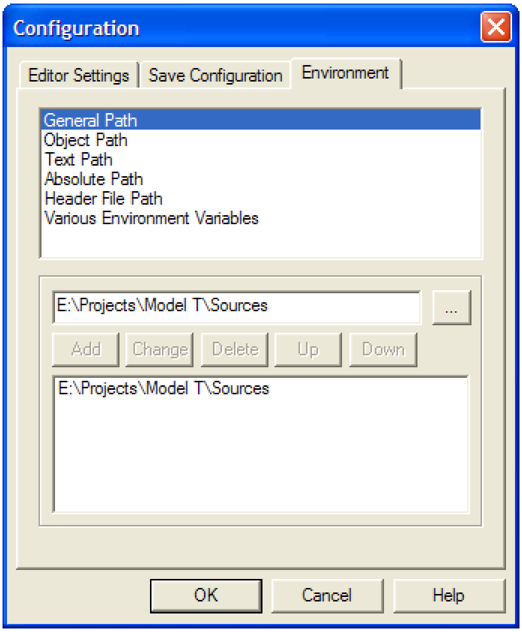
To fix this, select *File* > *Configuration*. The *Configuration* dialog box appears (Figure 1.44 on page 62).

Figure 1.44 Browsing for the Sources folder



Select the *Environment* tab and then *General Path*. Press the "..." button and navigate in the *Browse for Folder* dialog box for the folder that contains the `derivative.inc` file - the aforementioned `Sources` folder in the project directory. Press *OK* to close the *Browse for Folder* dialog box. The *Configuration* dialog box is now again active (Figure 1.45 on page 63).

Figure 1.45 Adding a GENPATH



E:\Projects\Model T\Sources

E:\Projects\Model T\Sources

Press the *Add* button, and the path to the `derivative.inc` file "*E:\Projects\Model T\Sources*" now appears in the lower panel. Press *OK*. An asterisk now appears in the Title bar, so save the change to the configuration by pressing the *Save* button in the Toolbar or by selecting *File > Save Configuration*. The asterisk disappears.

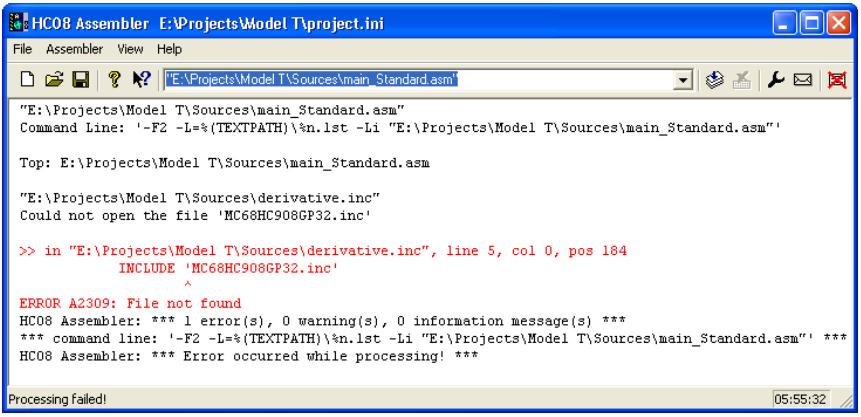
TIP You can clear the messages in the Assembler window at any time by selecting *View > Log > Clear Log*.

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

Select *File > Assemble* and again navigate to the *`main_Standard.asm`* file and press *Open*. However, the *A2309* error message reappears but this time for a different include file - `MC68HC908GP32.inc`. (Figure 1.46 on page 64).

NOTE In this case, the derivative.inc file has this statement: `INCLUDE 'MC68HC908GP32.inc'`. Therefore, a prior reading of the assembly-code and include files would suggest these include files might require `GENPATH` configurations. Therefore, you should set any needed GENPATH in advance of assembling the source-code files.

Figure 1.46 Assemble attempt #2


HC08 Assembler E:\Projects\Model T\project.ini
File Assembler View Help
"E:\Projects\Model T\Sources\main_Standard.asm"

```
"E:\Projects\Model T\Sources\main_Standard.asm"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"'

Top: E:\Projects\Model T\Sources\main_Standard.asm

"E:\Projects\Model T\Sources\derivative.inc"
Could not open the file 'MC68HC908GP32.inc'

>> in "E:\Projects\Model T\Sources\derivative.inc", line 5, col 0, pos 184
            INCLUDE 'MC68HC908GP32.inc'
                     ^
ERROR A2309: File not found
HC08 Assembler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"' ***
HC08 Assembler: *** Error occurred while processing! ***
```

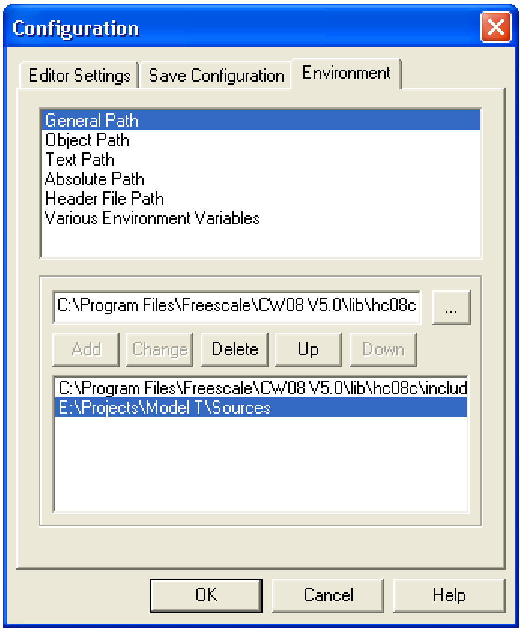
Processing failed!   05:55:32

You fix this by repeating the `GENPATH` routine for the other include file (Figure 1.47 on page 65). The `MC68HC908GP32.inc` file is located at this path:

`CW08 V5.x\lib\c08c\include`

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

Figure 1.47 Adding another GENPATH

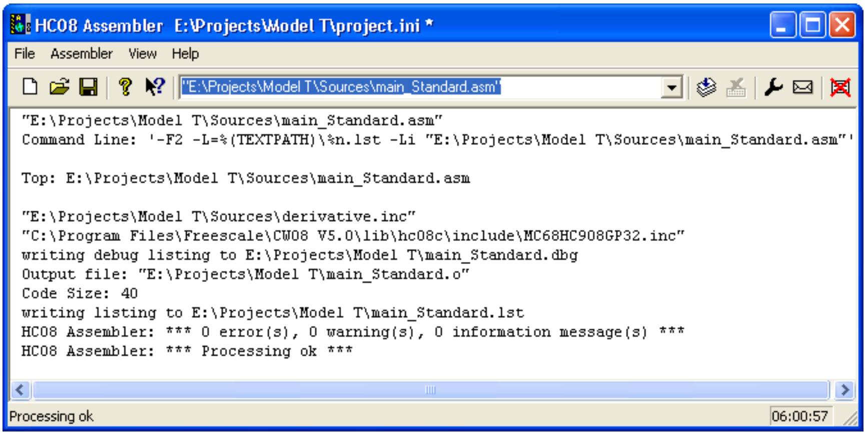


C:\Program Files\Freescale\CW08 V5.0\lib\hc08c

C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\includ
E:\Projects\Model T\Sources

After the `GENPATH` is set up for the second include file and saved as before, you can try to assemble the `main_Standard.asm` file for the third time (Figure 1.48 on page 65).

Figure 1.48 Assemble attempt #3 - success!

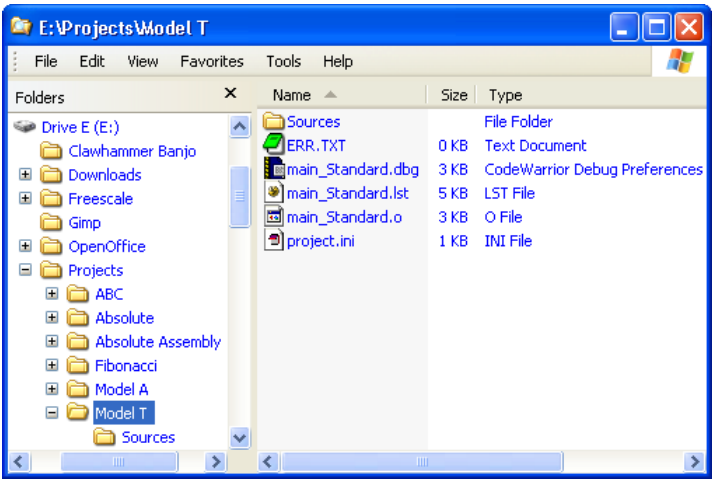

HC08 Assembler E:\Projects\Model T\project.ini *
File
Assembler
View
Help
"E:\Projects\Model T\Sources\main_Standard.asm"
Processing ok
06:00:57

```
"E:\Projects\Model T\Sources\main_Standard.asm"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main_Standard.asm"'

Top: E:\Projects\Model T\Sources\main_Standard.asm

"E:\Projects\Model T\Sources\derivative.inc"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\MC68HC908GP32.inc"
writing debug listing to E:\Projects\Model T\main_Standard.dbg
Output file: "E:\Projects\Model T\main_Standard.o"
Code Size: 40
writing listing to E:\Projects\Model T\main_Standard.lst
HC08 Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC08 Assembler: *** Processing ok ***
```

The Macro Assembler indicates successful assembling and indicated that the Code Size was 40 bytes. The message "`*** 0 error(s),`" indicates that the `main_Standard.asm` file assembled without errors. Do not forget to save the configuration one additional time.

The Assembler also generated a `main_Standard.dbg` file (for use with the Simulator/ Debugger), a `main_Standard.o` object file (for further processing with the Linker), and a `main_Standard.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_Standard.o`. The debug file has the same name as the input module, but with the '`*.dbg`' extension - `main_Standard.dbg` and the assembly output listing file has the `*.lst` extension (Figure 1.49 on page 66).

Figure 1.49 Project directory after a successful assembly



Name	Size	Type
Sources		File Folder
ERR.TXT	0 KB	Text Document
main_Standard.dbg	3 KB	CodeWarrior Debug Preferences
main_Standard.lst	5 KB	LST File
main_Standard.o	3 KB	O File
project.ini	1 KB	INI File

The `ERR.TXT` file is present in the project directory on account of the earlier failed attempts at assembling. The `ERR.TXT` file is empty upon a successful assembly. You can delete this file. Let's take an additional look at the `project.ini` file (Figure 1.8 on page 25).

Listing 1.9 project.ini file after GENPATH environmental variable is created

```
[AHC08_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,66,87,767,535
EditorType=1
Options=-F2 -L=%(TEXTPATH)\%n.lst -Li
RecentCommandLine0=""E:\Projects\Model T\Sources\main_Standard.asm""
CurrentCommandLine=""E:\Projects\Model T\Sources\main_Standard.asm""
[Environment Variables]
GENPATH=C:\Program Files\Freescale\CW08 V5.x\
lib\hc08c\include;E:\Projects\Model T\Sources
OBJPATH=
TEXTPATH=
ABSPATH=
LIBPATH=
```

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

Linking the application

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

Linking with CodeWarrior

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

PRM file

If you are using CodeWarrior to manage your project, a pre-configured PRM file for a particular derivative is already set up (Listing 1.10 on page 67). Listing 1.11 on page 68 is an example Linker PRM file for the RS08 derivative.

Listing 1.10 Linker PRM file for the GP32 derivative - Project.prm

```
/* This is a linker parameter file for the GP32 */

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. */
      ROM                     =  READ_ONLY    0x8000 TO 0xFDFF;
      Z_RAM                   =  READ_WRITE   0x0040 TO 0x00FF;
      RAM                     =  READ_WRITE   0x0100 TO 0x023F;
END

PLACEMENT /* Here all predefined and user segments are placed into the
             SEGMENTS defined above. */
      DEFAULT_RAM                        INTO  RAM;
      _DATA_ZEROPAGE, MY_ZEROPAGE       INTO  Z_RAM;
      DEFAULT_ROM, ROM_VAR, STRINGS     INTO  ROM;
END

STACKSIZE 0x50

//VECTOR 0 _Startup     /* Reset vector: This is the default entry
//                         point for a C/C++ application. */
//VECTOR 0 Entry  /* Reset vector: this is the default entry point
                     for an Assembly application. */
//INIT Entry      /* For assembly applications: that this is as
                     well the initialization entry point */
```

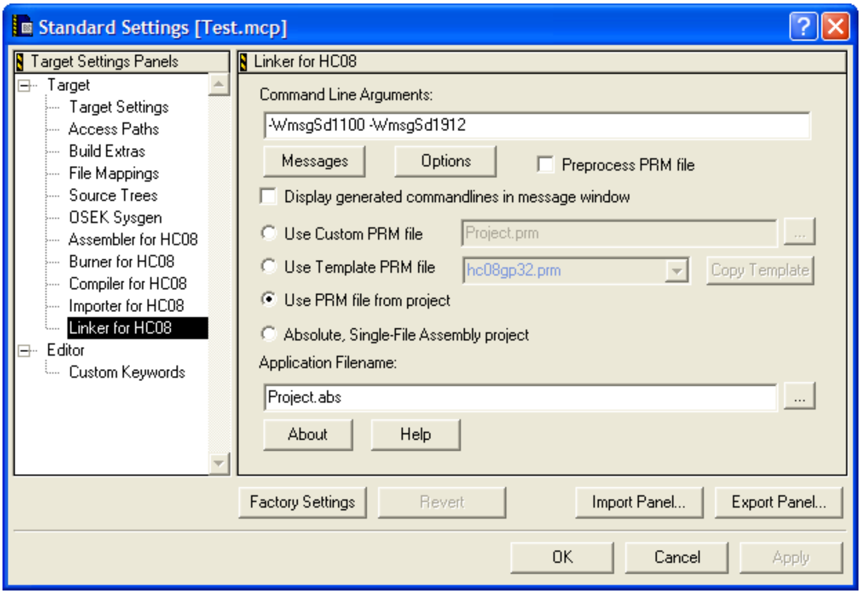
Listing 1.11 Linker PRM file for the RS08 derivative

```
LINK test_rs08.abs
NAMES test_rs08.o END
SEGMENTS
  TINY_RAM      = READ_WRITE 0x0000 TO 0x000D;
  DIRECT_RAM    = READ_WRITE 0x0020 TO 0x004F;
  ROM           = READ_ONLY  0x3800 TO 0x3FFB;
  RESET_JMP_AREA= READ_ONLY  0x3FFD TO 0x3FFF;
END

PLACEMENT
  DEFAULT_ROM                                         INTO ROM;
  DEFAULT_RAM                                         INTO DIRECT_RAM;

  TINY_RAM_VARS,                                      INTO TINY_RAM;
  DIRECT_RAM_VARS                                     INTO DIRECT_RAM,
TINY_RAM;
END

STACKSIZE 0x00 // no stack for RS08

VECTOR 0 Entry
INIT Entry
```

The Linker PRM file allocates memory for the stack and the sections named in the assembly source code files. If the sections in the source code are not specifically referenced in the `PLACEMENT` section, then these sections are included in `DEFAULT_ROM` or `DEFAULT_RAM`. You may use a different PRM file for each build target instead of the default PRM file generated by the Wizard - `Project.prm`.

The *Linker for HC08* preference panel controls which PRM file is used for your CodeWarrior project. The default PRM file for a CodeWarrior project is the PRM file in the project window. Let’s see what other options exist for the PRM file. From the *Edit* menu, select *<target_name> Settings... > Target > Linker for HC08*. The *Linker for HC08* preference panel appears (Figure 1.50 on page 69).

Figure 1.50 Linker for HC08 preference panel



Command Line Arguments: -WmsgSd1100 -WmsgSd1912

□ Preprocess PRM file

□ Display generated commandlines in message window

○ Use Custom PRM file: Project.prm

○ Use Template PRM file: hc08gp32.prm

● Use PRM file from project

○ Absolute, Single-File Assembly project

Application Filename: Project.abs

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

- *Use Custom PRM file*

  This option will browse for an existing PRM file for the build target.

- *Use Template PRM file*

  This option uses a template PRM in the pull-down menu and copies it for use in your build target.

- *Use PRM file from project* - the default

- *Absolute, Single-File Assembly project.*

  An absolute assembly project does not require a PRM file. Therefore, the configuration information that is otherwise present in a PRM file must be included in a single-file `*.asm` file. Only one `*.asm` file is allowed for absolute assembly.

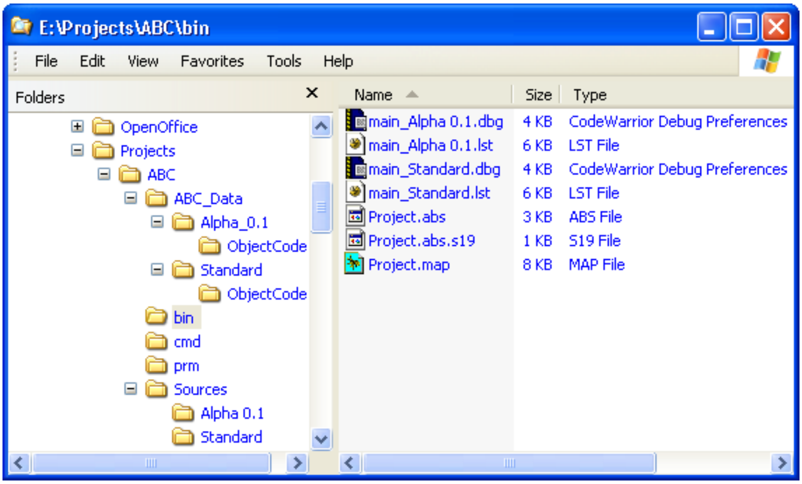
In case you want to change the filename of the application, you can determine the filename and its path with the *Application Filename:* text box.

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 1.10 on page 67.

Linking the object-code files

You can run this relocatable assembly project from the Project menu: Select *Project > Make* or *Project > Debug*. The Linker generates a `*.abs` file and a `*.abs.s19` standard S-Record File in the `bin` subfolder of the project directory. You can use an S-Record File to program ROM memory (Figure 1.51 on page 70).

Figure 1.51 bin folder in the project directory in Windows Explorer after linking

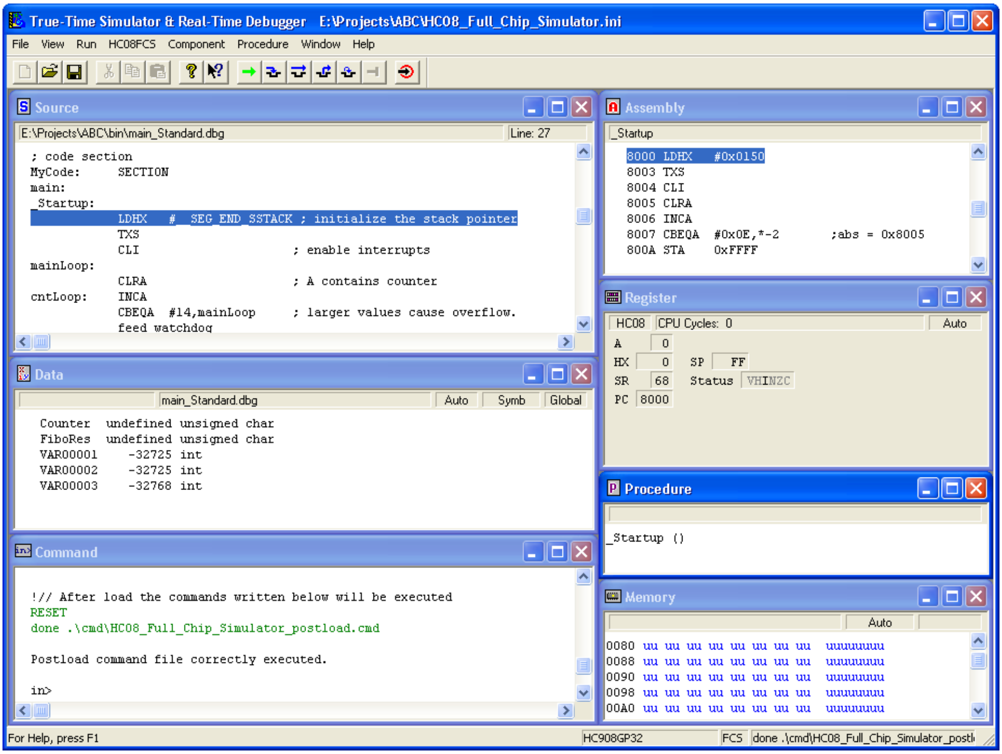


Name	Size	Type
main_Alpha 0.1.dbg	4 KB	CodeWarrior Debug Preferences
main_Alpha 0.1.lst	6 KB	LST File
main_Standard.dbg	4 KB	CodeWarrior Debug Preferences
main_Standard.lst	6 KB	LST File
Project.abs	3 KB	ABS File
Project.abs.s19	1 KB	S19 File
Project.map	8 KB	MAP File

The `Project.abs`, `Project.abs.s19`, and `Project.map` files in the Figure above are the Linker output files resulting from the object-code and PRM files and configuration in the build target that is selected in the *Targets* panel in the project window.

The *Full Chip Simulation* option in CodeWarrior was selected when the project was created, so if *Project > Debug* is selected, the debugger opens and you can follow each assembly-code instruction during the execution of the program with the Hiwave Simulator (Figure 1.52 on page 71).

Figure 1.52 hiwave.exe - Simulator/Debugger build tool


True-Time Simulator & Real-Time Debugger E:\Projects\ABC\HC08_Full_Chip_Simulator.ini
File View Run HC08FCS Component Procedure Window Help
Source
E:\Projects\ABC\bin\main_Standard.dbg
Line: 27
Assembly
_Startup
Register
HC08
CPU Cycles: 0
Auto
Data
main_Standard.dbg
Auto
Symb
Global
Procedure
Command
Memory
Auto
For Help, press F1
HC908GP32
FCS
done .\cmd\HC08_Full_Chip_Simulator_postl

```
; 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.
            feed watchdog
```

```
8000 LDHX   #0x0150
8003 TXS
8004 CLI
8005 CLRA
8006 INCA
8007 CBEQA  #0x0E,*-2        ;abs = 0x8005
800A STA    0xFFFF
```

```
A      0
HX     0   SP    FF
SR    68   Status VHINZC
PC  8000
```

```
 Counter  undefined unsigned char
 FiboRes  undefined unsigned char
 VAR00001    -32725 int
 VAR00002    -32725 int
 VAR00003    -32768 int
```

```
_Startup ()
```

```
!// After load the commands written below will be executed
RESET
done .\cmd\HC08_Full_Chip_Simulator_postload.cmd

Postload command file correctly executed.

in>
```

```
0080 uu uu uu uu uu uu uu uu  uuuuuuuu
0088 uu uu uu uu uu uu uu uu  uuuuuuuu
0090 uu uu uu uu uu uu uu uu  uuuuuuuu
0098 uu uu uu uu uu uu uu uu  uuuuuuuu
00A0 uu uu uu uu uu uu uu uu  uuuuuuuu
```

You can single-step the Simulator through the Fibonacci program from the *Run* menu in the Simulator (*Run* > *Assembly Step* or *Ctrl+F11*). You can monitor the seven panels in the Simulator while following the logic in the Fibonacci application.

Linking with the 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.

- Using a text editor, create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as `<target_name>.prm`.
- Store the PRM file in a convenient location, such as the project directory.
- In the `<project_name>.prm` file, change the name of the executable (`*.abs`) file to whatever you choose, e.g., `<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 an MC68HC908GP32 from another (CodeWarrior) project was adapted — is shown in Listing 1.12 on page 72.

Listing 1.12 Layout of a PRM file for the Linker - Model T.prm

```
/* This is a linker parameter file for the GP32 */

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

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in
            PLACEMENT below. */
      ROM                   =  READ_ONLY    0x8000 TO 0xFDFF;
      Z_RAM                 =  READ_WRITE   0x0040 TO 0x00FF;
      RAM                   =  READ_WRITE   0x0100 TO 0x023F;
END

PLACEMENT /* Here all predefined and user segments are placed into the
             SEGMENTS defined above. */
      DEFAULT_RAM                        INTO  RAM;
      _DATA_ZEROPAGE, MY_ZEROPAGE        INTO  Z_RAM;
      DEFAULT_ROM, ROM_VAR, STRINGS      INTO  ROM;
END

STACKSIZE 0x50

VECTOR 0 _Startup  /* Reset vector: this is the default entry point
                      for an Assembly application. */
INIT _Startup      /* For assembly applications: that this is as
                      well the initialization entry point */
```

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

NOTE The default size for the stack using the CodeWarrior Wizard for the GP32 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.

- Start the Linker.

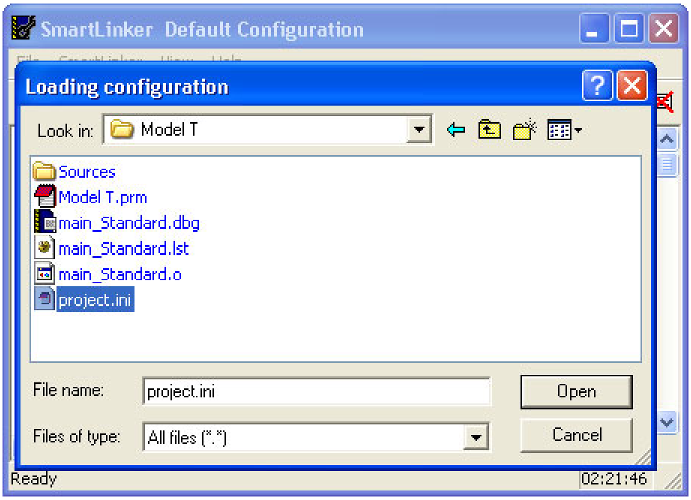
  The SmartLinker tool is located in the `prog` folder in the CodeWarrior installation: `prog\linker.exe`

- Press *Close* to close the *Tip of the Day* dialog box.

- 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:

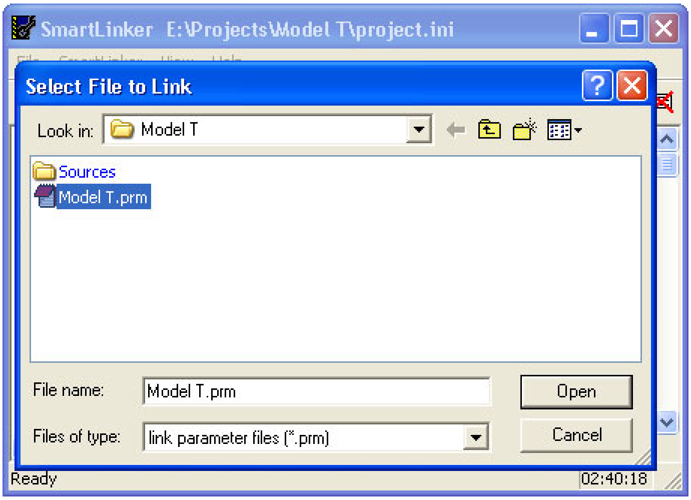
  File > Load Configuration and navigate to and select the project's configuration file (Figure 1.53 on page 73).

Figure 1.53 HC(S)08 Linker



- Press *Open* to load the configuration file. The project directory is now the current directory for the Linker. You should press the *Save* button to save the configuration. From the *File* menu in the Smart Linker, select *Link*: (*File > Link)*. The *Select File to Link* dialog box appears (Figure 1.54 on page 74).

Figure 1.54 Select File to Link dialog box



Look in: Model T

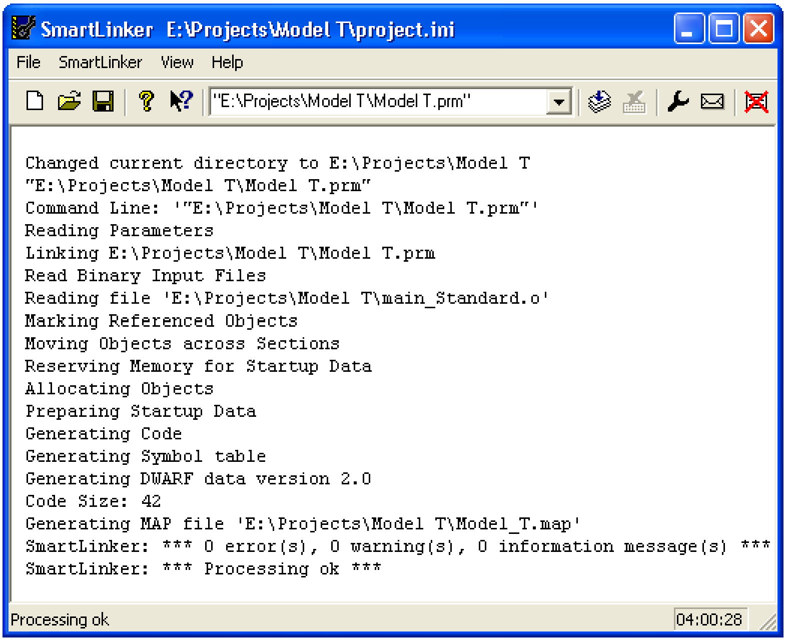
Model T.prm

File name: Model T.prm

Files of type: link parameter files (*.prm)

- Browse to locate and select the PRM file for your project. Press *Open*. The Smart Linker links the object-code files in the `NAMES` section to produce the executable `*.abs` file, as specified in the `LINK` portion of the Linker PRM file (Figure 1.55 on page 75).

Figure 1.55 Linker main window after linking



SmartLinker E:\Projects\Model T\project.ini
File SmartLinker View Help
"E:\Projects\Model T\Model T.prm"

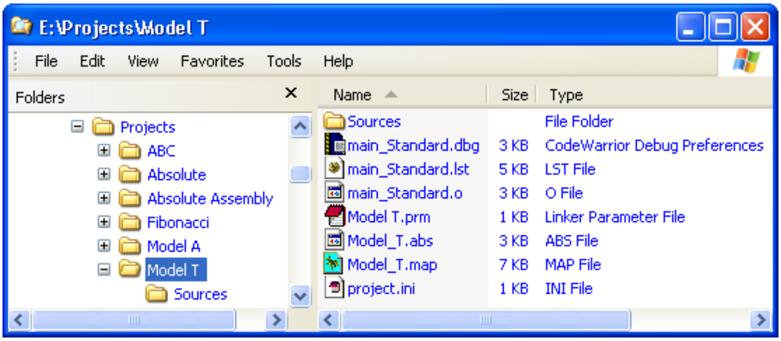
```
Changed current directory to E:\Projects\Model T
"E:\Projects\Model T\Model T.prm"
Command Line: '"E:\Projects\Model T\Model T.prm"'
Reading Parameters
Linking E:\Projects\Model T\Model T.prm
Read Binary Input Files
Reading file 'E:\Projects\Model T\main_Standard.o'
Marking Referenced Objects
Moving Objects across Sections
Reserving Memory for Startup Data
Allocating Objects
Preparing Startup Data
Generating Code
Generating Symbol table
Generating DWARF data version 2.0
Code Size: 42
Generating MAP file 'E:\Projects\Model T\Model_T.map'
SmartLinker: *** 0 error(s), 0 warning(s), 0 information message(s) ***
SmartLinker: *** Processing ok ***
```

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

- The current directory for the Linker is the project directory, `E:\Projects\Model T`.
- The `Model T.prm` file was used to name the executable file, which object files were linked, and how the RAM and ROM memory areas were allocated for the relocatable sections. The Reset and application entry points were also specified in this file.
- There was one object file, `main_Standard.o`.
- The output format was DWARF 2.0.
- The Code Size was 42 bytes.
- A Linker Map file - `Model_T.map` was generated.
- No errors or warnings occurred and no information messages were 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 1.56 on page 76 shows the contents of the project directory after the relocatable assembly project was linked.

Figure 1.56 Project directory after linking

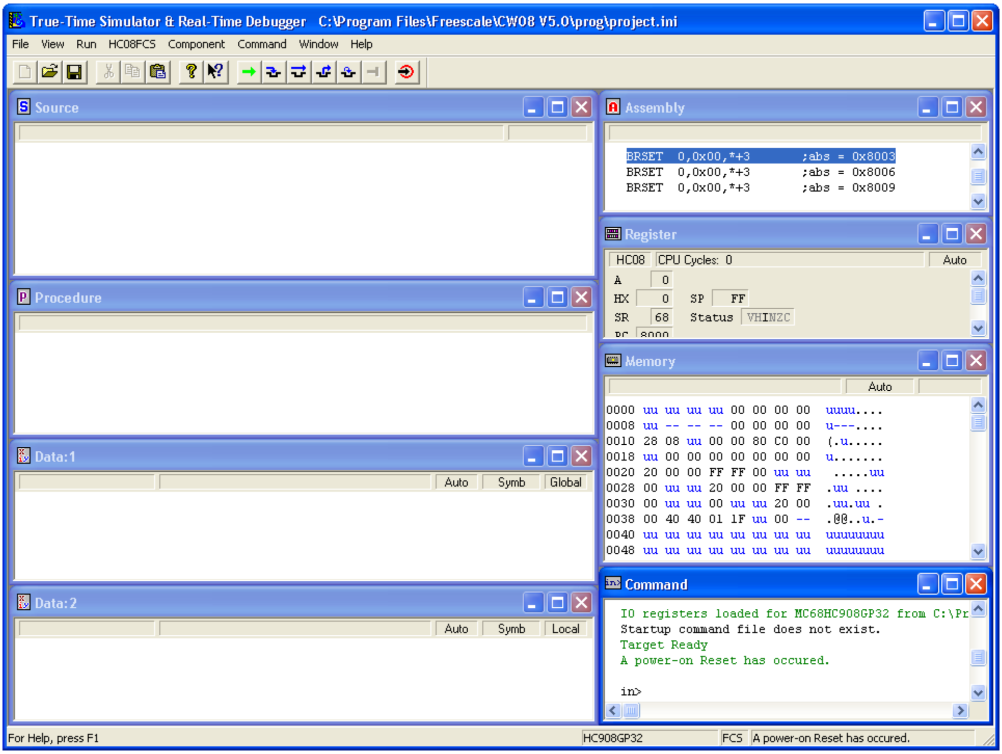


Name	Size	Type
Sources		File Folder
main_Standard.dbg	3 KB	CodeWarrior Debug Preferences
main_Standard.lst	5 KB	LST File
main_Standard.o	3 KB	O File
Model T.prm	1 KB	Linker Parameter File
Model_T.abs	3 KB	ABS File
Model_T.map	7 KB	MAP File
project.ini	1 KB	INI File

The Simulator/Debugger Build Tool, `hiwave.exe`, located in the `prog` folder in the CodeWarrior installation could be used to simulate the program that was assembled using the `main_Standard.asm` source-code file and linked to generate the `Model_T.abs` executable.

Start the Simulator. The GUI for the Simulator appears (Figure 1.57 on page 77).

Figure 1.57 True-Time Simulator & Real-Time Debugger


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\CW08 V5.0\prog\project.ini
File View Run HC08FCS Component Command Window Help
Source
Procedure
Data:1
Auto Symb Global
Data:2
Auto Symb Local
Assembly
Register
Memory
Auto
Command
For Help, press F1

```
BRSET   0,0x00,*+3        ;abs = 0x8003
BRSET   0,0x00,*+3        ;abs = 0x8006
BRSET   0,0x00,*+3        ;abs = 0x8009
```

```
HC08  CPU Cycles: 0                Auto
A      0
HX     0    SP     FF
SR    68    Status VHINZC
PC  8000
```

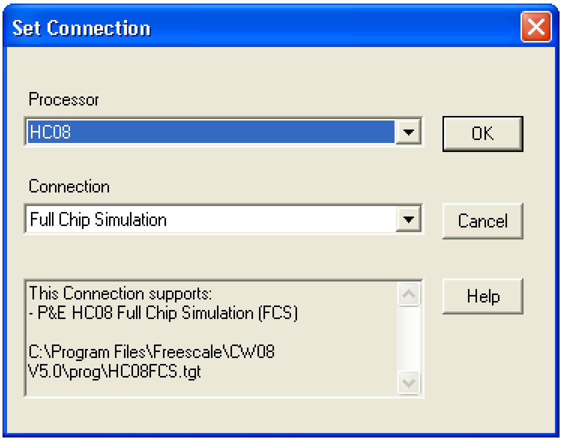
```
0000 uu uu uu uu 00 00 00 00  uuuu....
0008 uu -- -- -- 00 00 00 00  u---....
0010 28 08 uu 00 00 80 C0 00  (.u.....
0018 uu 00 00 00 00 00 00 00  u.......
0020 20 00 00 FF FF 00 uu uu   .....uu
0028 00 uu uu 20 00 00 FF FF  .uu ....
0030 00 uu uu 00 uu uu 20 00  .uu.uu .
0038 00 40 40 01 1F uu 00 --  .@@..u.-
0040 uu uu uu uu uu uu uu uu  uuuuuuuu
0048 uu uu uu uu uu uu uu uu  uuuuuuuu
```

```
IO registers loaded for MC68HC908GP32 from C:\Pr
Startup command file does not exist.
Target Ready
A power-on Reset has occured.

in>
```

```
HC908GP32    FCS    A power-on Reset has occured.
```

Select *Set Connection...* from the *Component* menu. The *Set Connection* dialog box appears (Figure 1.58 on page 78)

Figure 1.58 Set Connection dialog box



Processor: HC08

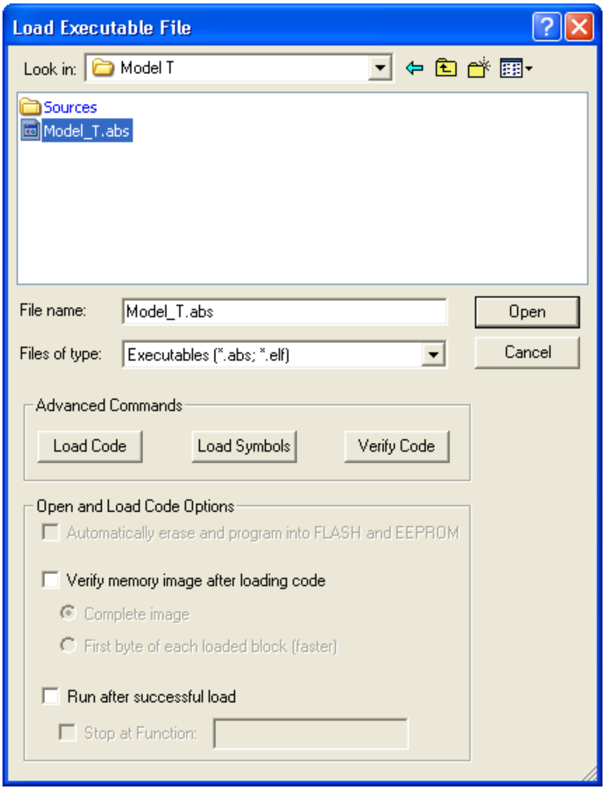
Connection: Full Chip Simulation

This Connection supports:
- P&E HC08 Full Chip Simulation (FCS)

C:\Program Files\Freescale\CW08 V5.0\prog\HC08FCS.tgt

The CPU derivative for this project is in the HC08 subfamily, so select *HC08* from the *Processor* pull-down menu. Select *Full Chip Simulation* in the *Connection* pull-down menu. Press *OK*. From the `File` menu, select *Load Application...*. The *Load Executable File* dialog box appears (Figure 1.59 on page 79)

Figure 1.59 Load Executable File dialog box



Look in: Model T

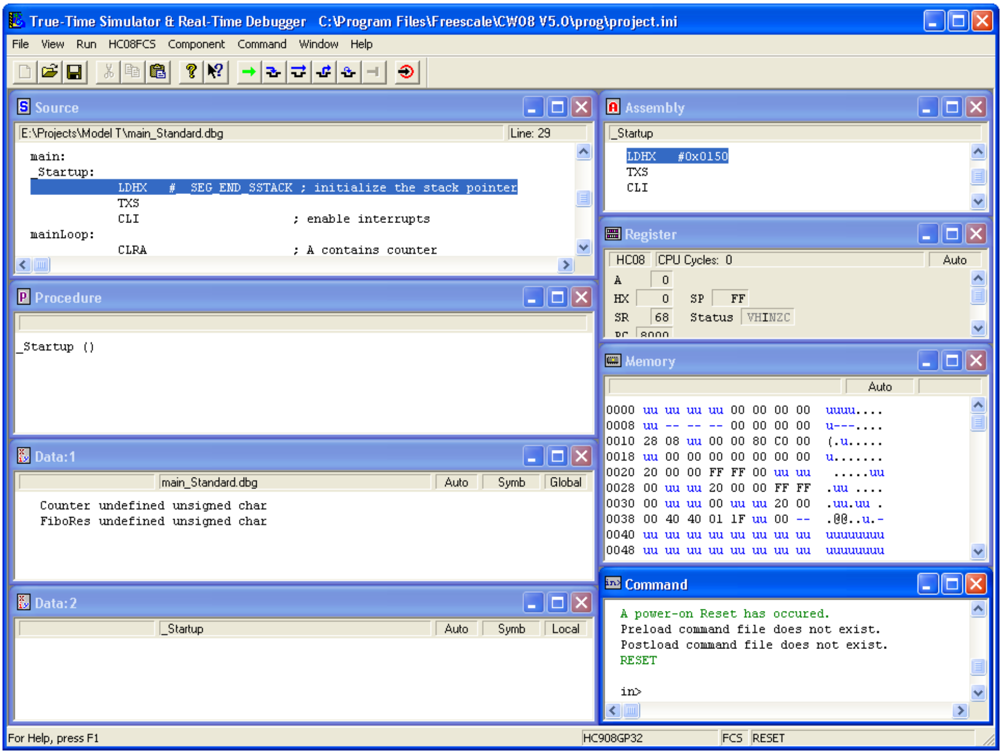
File name: Model_T.abs

Files of type: Executables (*.abs; *.elf)

- □ Automatically erase and program into FLASH and EEPROM
- □ Verify memory image after loading code
  - ● Complete image
  - ○ First byte of each loaded block (faster)
- □ Run after successful load
  - □ Stop at Function:

Browse to and select the `Model_T.abs` file in the project directory. Press *Open*. The Simulator is now set up to be run (Figure 1.60 on page 80).

Figure 1.60 Simulator is now ready


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\CW08 V5.0\prog\project.ini
File View Run HC08FCS Component Command Window Help
Source
Assembly
Register
Procedure
Memory
Data:1
Data:2
Command
Auto
Symb
Global
Local
For Help, press F1
HC908GP32
FCS
RESET

Source: E:\Projects\Model T\main_Standard.dbg Line: 29

```
main:
_Startup:
            LDHX   #__SEG_END_SSTACK ; initialize the stack pointer
            TXS
            CLI                         ; enable interrupts
mainLoop:
            CLRA                        ; A contains counter
```

Assembly: _Startup

```
LDHX    #0x0150
TXS
CLI
```

Register: HC08 CPU Cycles: 0

```
A     0
HX    0   SP    FF
SR   68   Status VHINZC
PC 8000
```

Memory:

```
0000 uu uu uu uu 00 00 00 00  uuuu....
0008 uu -- -- -- 00 00 00 00  u---....
0010 28 08 uu 00 00 80 C0 00  (.u.....
0018 uu 00 00 00 00 00 00 00  u.......
0020 20 00 00 FF FF 00 uu uu   .....uu
0028 00 uu uu 20 00 00 FF FF  .uu ....
0030 00 uu uu 00 uu uu 20 00  .uu.uu .
0038 00 40 40 01 1F uu 00 --  .@@..u.-
0040 uu uu uu uu uu uu uu uu  uuuuuuuu
0048 uu uu uu uu uu uu uu uu  uuuuuuuu
```

Procedure:

```
_Startup ()
```

Data:1: main_Standard.dbg

```
Counter undefined unsigned char
FiboRes undefined unsigned char
```

Data:2: _Startup

Command:

```
A power-on Reset has occured.
Preload command file does not exist.
Postload command file does not exist.
RESET

in>
```

You can repeatedly press the *Assembly Step (Ctrl+F11)* icon to single-step the Simulator through the assembly source-code and monitor the program’s logic of the Fibonacci application in the eight panels within the Simulator’s GUI.

Directly generating an ABS file

You can also use CodeWarrior or the Assembler build tool 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 CodeWarrior or the Assembler 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.

Using CodeWarrior to generate an ABS file

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

- No `PRM` file is required.
- The memory area allocation is determined directly in a single `*.asm` file assembly source file.
- CodeWarrior needs some configurations to be applied to the Linker and Assembler preference panels.

Start the CodeWarrior Wizard and create an assembler project in the usual manner. See "The Wizard" on page 19. Next, convert the `main_Standard.asm` relocatable assembly file to the absolute assembly file below in Listing 1.13 on page 81.

Adapting the main_Standard.asm file produced by the Wizard

Changing the `SECTION` directives in a relocatable assembly file to `ORG` directives is required. The `ORG` directives must specify the absolute memory areas for ROM and RAM. Listing 1.13 on page 81 is an adaptation of the `main_Standard.asm` file produced previously by the Wizard. This file may be used by CodeWarrior or the Assembler build tool.

Listing 1.13 Example source file — main_Standard.asm

```
;**************************************************************
;* This stationery serves as the framework for a user        *
;* application. For a more comprehensive program that        *
;* demonstrates the more advanced functionality of this      *
;* processor, please see the demonstration applications      *
;* located in the examples subdirectory of Codewarrior       *
;* for the HC08 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.
            STA   COPCTL            ; Feed the 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 1.14 on page 83 is a similar example for RS08.

Listing 1.14 Example source file abstest_rs08.asm

```
ABSENTRY entry; Specifies the application Entry point
XDEF entry ; Make the symbol entry visible (needed for debugging)
        ORG $40 ; Define an absolute constant section
var1: DC.B 5 ; Assign 5 to the symbol var1
        ORG $80 ; Define an absolute data section
data: DS.B 1 ; Define one byte variable in RAM at $80
        ORG $B00 ; 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 1.15 on page 83).

Listing 1.15 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 1.16 on page 84).

Listing 1.16 Using ABSENTRY to enter the entry-point address

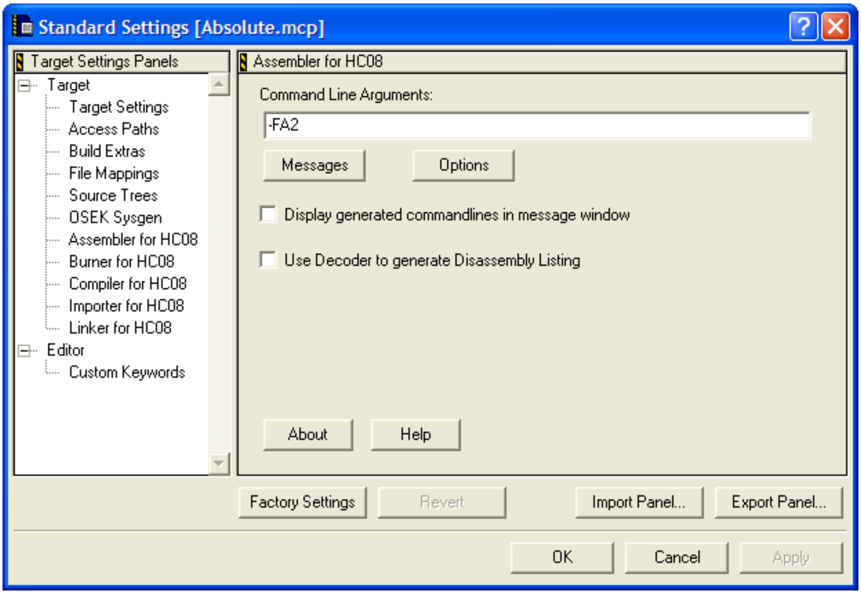
```
        ABSENTRY _Startup
```

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.

Reconfiguring CodeWarrior

From the *Edit* menu, open the *Assembler for HC08* preference panel. Select *Edit > <target_name> Settings... > Target > Assembler for HC08*. The *Assembler* preference panel appears (Figure 1.61 on page 84)

Figure 1.61 Assembler for HC08 preference panel



Standard Settings [Absolute.mcp]

Assembler for HC08

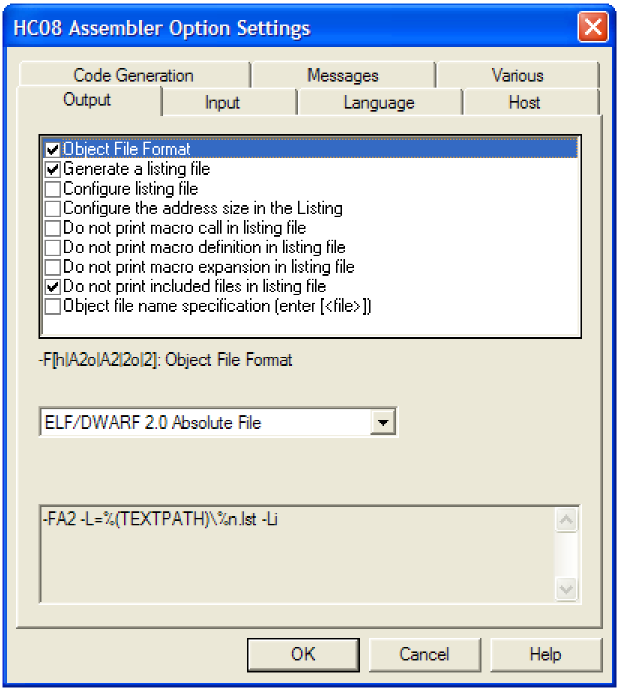
Command Line Arguments: -FA2

□ Display generated commandlines in message window

□ Use Decoder to generate Disassembly Listing

Press the *Options* button. The *HC08 Assembler Option Settings* dialog box appears (Figure 1.62 on page 85).

Figure 1.62 HC08 Assembler Option Settings dialog box



☑ Object File Format
☑ Generate a listing file
□ Configure listing file
□ Configure the address size in the Listing
□ Do not print macro call in listing file
□ Do not print macro definition in listing file
□ Do not print macro expansion in listing file
☑ Do not print included files in listing file
□ Object file name specification (enter [<file>])

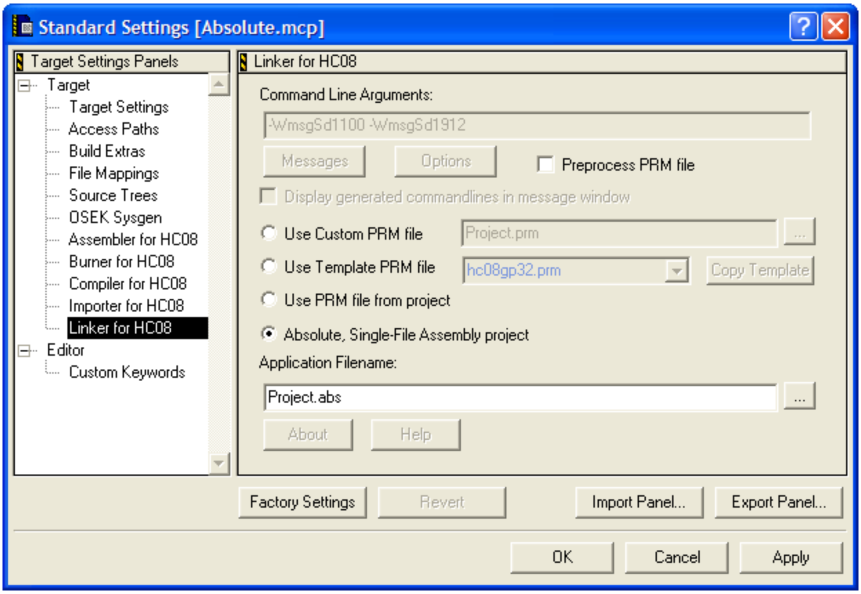
-F[h|A2o|A2|2o|2]: Object File Format

ELF/DWARF 2.0 Absolute File

```
-FA2 -L=%(TEXTPATH)\%n.lst -Li
```

In the *Output* panel, select *Object File Format > ELF/DWARF 2.0 Absolute File*. Press *OK* to close the dialog box. Now, select the *Linker for HC08* preference panel and select *Options*. The *Linker for HC08* preference panel opens (Figure 1.63 on page 86).

Figure 1.63 Linker for HC08 preference panel



Standard Settings [Absolute.mcp]

Linker for HC08

Command Line Arguments: -WmsgSd1100 -WmsgSd1912

□ Preprocess PRM file

□ Display generated commandlines in message window

○ Use Custom PRM file: Project.prm

○ Use Template PRM file: hc08gp32.prm

○ Use PRM file from project

● Absolute, Single-File Assembly project

Application Filename: Project.abs

Select the *Absolute, Single-File Assembly project* radio button and press *OK*. The assembler is now configured to directly produce an absolute assembly **.abs* output file.

Assembling and generating the application

All that is needed to produce the executable `*.abs` file is to select *Project > Make* or *Project > Debug*. CodeWarrior produces the same `*.abs` and `*.abs.s19` output files that the Assembler and Linker generated for relocatable assembly.

The `*.abs.s19` file generated in the `bin` subfolder of the project directory is a standard S-Record File. You can burn this file directly into a ROM memory.

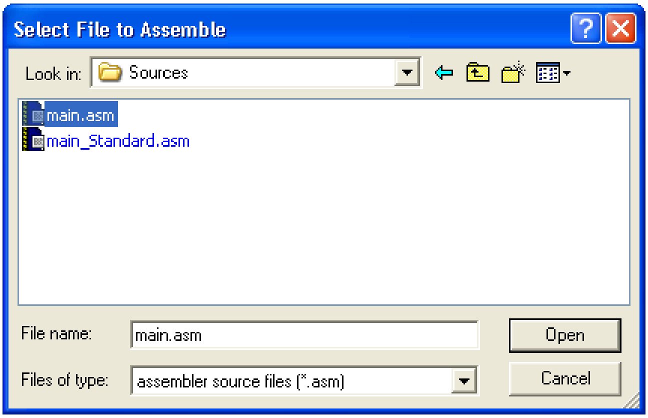
If you selected *Project > Debug*, the debugger opens and you can follow the execution of the program while assemble-stepping the Simulator. You can single-step the simulator through the program from the *Run* menu in the Simulator (*Run > Assembly Step* or *Ctrl + F11*).

Using the Assembler build tool for absolute assembly

Use the same project - Model T that was used for the relocatable assembly project. Use an absolute assembly source file of the type listed in Listing 1.13 on page 81, name the file - `main.asm`, and insert this file into the `Sources` file in the project directory.

1. Start the Assembler. You can do this by opening the `ahc08.exe` file in the `prog` folder in the HC08 CodeWarrior installation. The Assembler opens. Close the *Tip of the Day* dialog box.
2. Using *File > Load Configuration*, browse for project directory and set it to be the current directory for the Assembler.
3. Select *Assembler > Options*. The *Option Settings* dialog box appears.
4. In the *Output* dialog box, select the check box in front of the label *Object File Format*. The Assembler displays more information at the bottom of the dialog box.
5. Select the *ELF/DWARF 2.0 Absolute File* menuitem in the pull-down menu. Click *OK*.
6. Select the assembly source-code file that will be assembled: Select *File > Assemble*. The *Select File to Assemble* dialog box appears (Figure 1.64 on page 87).

Figure 1.64 Select File to Assemble dialog box

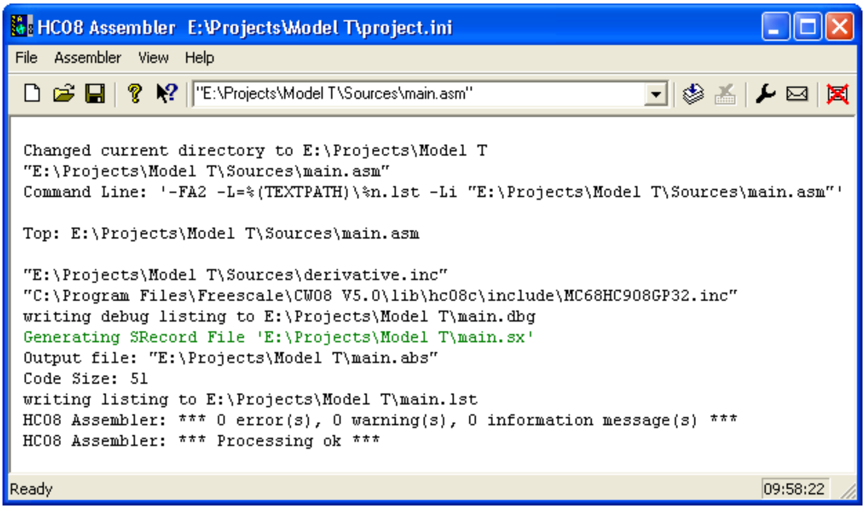


File name: main.asm

Files of type: assembler source files (*.asm)

7. Browse to the absolute-assembly source-code file - `main.asm`. 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 created in the assembler main window (Figure 1.65 on page 88).

Figure 1.65 Successful absolute assembly


HC08 Assembler E:\Projects\Model T\project.ini
File Assembler View Help
"E:\Projects\Model T\Sources\main.asm"
Ready
09:58:22

```
Changed current directory to E:\Projects\Model T
"E:\Projects\Model T\Sources\main.asm"
Command Line: '-FA2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main.asm"'

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

"E:\Projects\Model T\Sources\derivative.inc"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\MC68HC908GP32.inc"
writing debug listing to E:\Projects\Model T\main.dbg
Generating SRecord File 'E:\Projects\Model T\main.sx'
Output file: "E:\Projects\Model T\main.abs"
Code Size: 51
writing listing to E:\Projects\Model T\main.lst
HC08 Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC08 Assembler: *** Processing ok ***
```

The messages indicate that:

- An assembly source code (`main.asm`) file, plus `derivative.inc` and `MC68HC908GP32.inc` files were read as input.
- A debugging (`main.dbg`) file was generated in the project directory.
- An S-Record File was created, `main.sx`. This file can be used to program ROM memory.
- An absolute executable file was generated, `main.abs`.
- The Code Size is 51 bytes.
- An assembly outlet listing file (`main.lst`) was written to the project directory.

The `main.abs` file can be used as input to the Simulator, with which you can follow the execution of your program.

2

Assembler Graphical User Interface

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

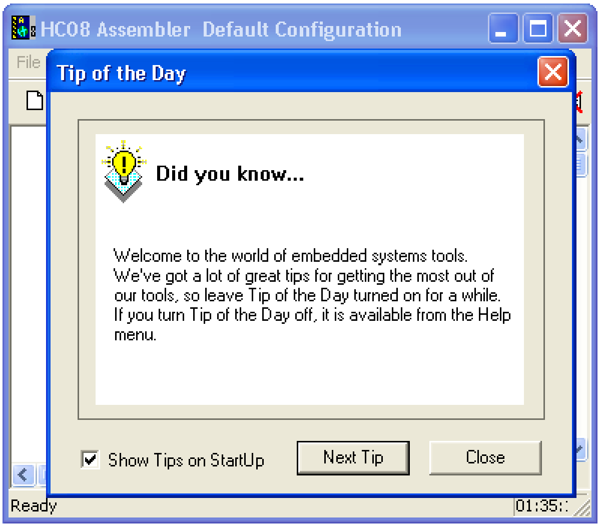
This chapter covers the following topics:

- Starting the Assembler on page 89
- Assembler Main Window on page 90
- Editor Setting dialog box on page 96
- Save Configuration dialog box on page 102
- Option Settings dialog box on page 105
- Message settings dialog box on page 106
- About... dialog box on page 110
- Specifying the input file on page 110
- Message/Error feedback on page 111

Starting the Assembler

When you start the Assembler, the Assembler displays a standard *Tip of the Day* (Figure 2.1 on page 90) window containing news and tips about the Assembler.

Figure 2.1 Tip of the Day dialog box



Welcome to the world of embedded systems tools. We've got a lot of great tips for getting the most out of our tools, so leave Tip of the Day turned on for a while. If you turn Tip of the Day off, it is available from the Help menu.

Click *Next Tip* to see the next piece of information about the Assembler.

Click *Close* to close the *Tip of the Day* dialog box.

If you do not want the Assembler to automatically open the standard *Tip of the Day* window when the Assembler is started, uncheck *Show Tips on StartUp*.

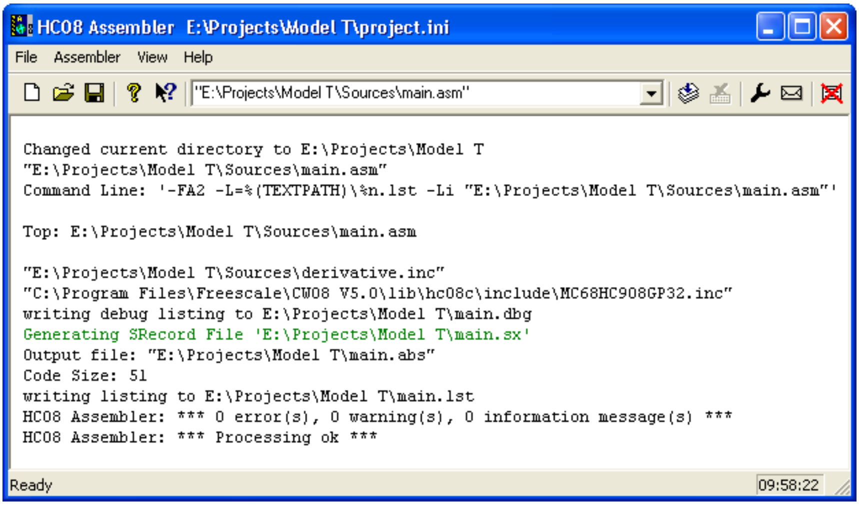
If you want the Assembler to automatically open the standard *Tip of the Day* window at Assembler start up, choose *Help* > *Tip of the Day*.... The Assembler displays the *Tip of the Day* dialog box. Check the *Show Tips on StartUp* check box.

Assembler Main Window

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

The assembler window consists of a window title, a menu bar, a toolbar, a content area, and a status bar (Figure 2.2 on page 91).

Figure 2.2 HC08 Assembler main window


HC08 Assembler E:\Projects\Model T\project.ini
File
Assembler
View
Help
"E:\Projects\Model T\Sources\main.asm"
Ready
09:58:22

```
Changed current directory to E:\Projects\Model T
"E:\Projects\Model T\Sources\main.asm"
Command Line: '-FA2 -L=%(TEXTPATH)\%n.lst -Li "E:\Projects\Model T\Sources\main.asm"'

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

"E:\Projects\Model T\Sources\derivative.inc"
"C:\Program Files\Freescale\CW08 V5.0\lib\hc08c\include\MC68HC908GP32.inc"
writing debug listing to E:\Projects\Model T\main.dbg
Generating SRecord File 'E:\Projects\Model T\main.sx'
Output file: "E:\Projects\Model T\main.abs"
Code Size: 51
writing listing to E:\Projects\Model T\main.lst
HC08 Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC08 Assembler: *** Processing ok ***
```

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 on page 96Editor Setting dialog box on page 96 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.

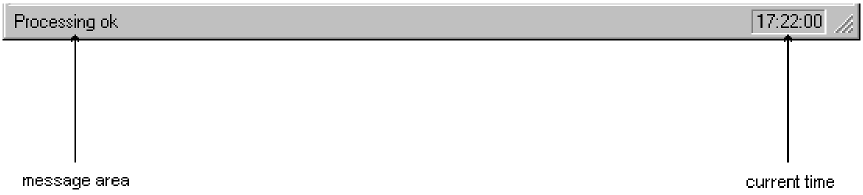
Pressing the *Message Dialog Box* button ✉ opens the *Message Settings* dialog box.

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

Status bar

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

Figure 2.3 Status bar



Processing ok

17:22:00

Assembler menu bar

The following menus are available in the menu bar (Table 2.1 on page 93):

Table 2.1 Menu bar options

Menu	Description
File menu on page 93	Contains entries to manage Assembler configuration files
Assembler menu on page 95	Contains entries to set Assembler options
View menu on page 95	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 on page 96.),
- the *Tips of the Day* settings, including its startup configuration, and what is the current entry, and
- Configuration files are text files which have the standard `*.ini` extension. You can define as many configuration files as required for the project and can switch among the different configuration files using the *File > Load Configuration, File | Save Configuration* menu entries, or the corresponding toolbar buttons.

Table 2.2 File menu options

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

Table 2.2 File menu options (*continued*)

Menu entry	Description
Configuration...	Opens the *Configuration* dialog box to specify the editor used for error feedback and which parts to save with a configuration. See Editor Setting dialog box on page 96 and Save Configuration dialog box on page 102.
1. .... project.ini 2.	Recent project list. This list can be used to reopen a recently opened project.
Exit	Closes the Assembler.

Assembler menu

The Assembler menu (Table 2.3 on page 95) allows you to customize the Assembler. You can graphically set or reset the Assembler options or to stop the assembling process.

Table 2.3 Assembler menu options

Menu entry	Description
Options	Defines the options which must be activated when assembling an input file. (See Option Settings dialog box on page 105)
Messages	Maps messages to a different message class (See Message settings dialog box on page 106)
Stop assembling	Stops the assembling of the current source file.

View menu

The View menu (Table 2.4 on page 95) lets you customize the assembler window. You can specify if the status bar or the toolbar must be displayed or be hidden. You can also define the font used in the window or clear the window.

Table 2.4 View menu options

Menu entry	Description
Toolbar	Switches display from the toolbar in the assembler window.
Status Bar	Switches display from the status bar in the assembler window.

Table 2.4 View menu options (*continued*)

Menu entry	Description
Log...	Customizes the output in the assembler window content area. The following two entries in this table are available when Log... is selected:
Change Font	Opens a standard font dialog box. The options selected in the font dialog box are applied to the assembler window content area.
Clear Log	Clears the assembler window content area.

Editor 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.

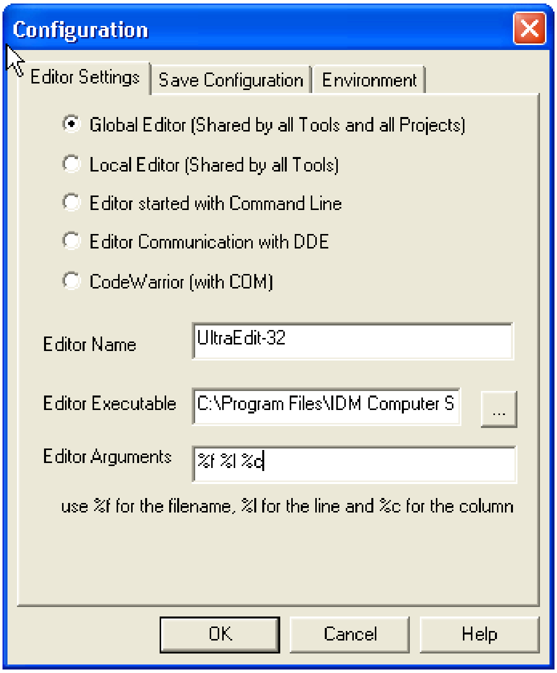
These are the main entries for the Editor configuration:

- "Global Editor (shared by all tools and projects)" on page 96
- "Local Editor (shared by all tools)" on page 97
- "Editor started with the command line" on page 98
- "Editor started with DDE" on page 99
- "CodeWarrior with COM" on page 100

Global Editor (shared by all tools and projects)

This entry (Figure 2.4 on page 97) is shared by all tools (Compiler/Linker/Assembler/...) for all projects. This setting is stored in the `[Editor]` section of the `mcutools.ini` global initialization file. Some Modifiers on page 101 can be specified in the editor command line.

Figure 2.4 Global Editor Configuration dialog box



Configuration

● Global Editor (Shared by all Tools and all Projects)

○ Local Editor (Shared by all Tools)

○ Editor started with Command Line

○ Editor Communication with DDE

○ CodeWarrior (with COM)

Editor Name: UltraEdit-32

Editor Executable: C:\Program Files\IDM Computer S

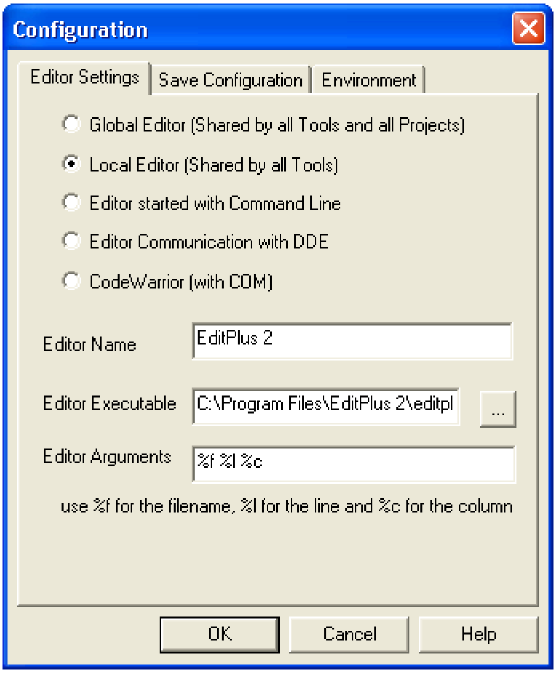
Editor Arguments: %f %l %c

use %f for the filename, %l for the line and %c for the column

Local Editor (shared by all tools)

This entry (on page 98) is shared by all tools (Compiler, Linker, Assembler, ...) for the current project. This setting is stored in the `[Editor]` section of the local initialization file, usually `project.ini` in the current directory. Some Modifiers on page 101 can be specified in the editor command line.

Figure 2.5 Local editor configuration dialog box



○ Global Editor (Shared by all Tools and all Projects)
● Local Editor (Shared by all Tools)
○ Editor started with Command Line
○ Editor Communication with DDE
○ CodeWarrior (with COM)

Editor Name: EditPlus 2

Editor Executable: C:\Program Files\EditPlus 2\editpl

Editor Arguments: %f %l %c

use %f for the filename, %l for the line and %c for the column

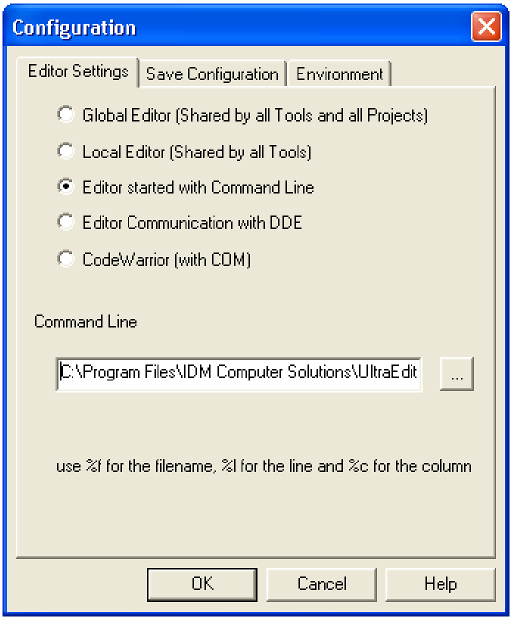
Editor started with the command line

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

Enter the command which should be used to start the editor (Figure 2.6 on page 99).

The format from the editor command depends on the syntax which should be used to start the editor. Modifiers can be specified in the editor command line to refer to a filename and line and column position numbers. (See the Modifiers on page 101 section below.)

Figure 2.6 Command-Line Editor configuration



○ Global Editor (Shared by all Tools and all Projects)

○ Local Editor (Shared by all Tools)

● Editor started with Command Line

○ Editor Communication with DDE

○ CodeWarrior (with COM)

Command Line

C:\Program Files\IDM Computer Solutions\UltraEdit

use %f for the filename, %l for the line and %c for the column

Example of configuring a command-line editor

The following case portrays the syntax used for configuring an external editors. Listing 2.1 on page 99 can be used for the UltraEdit-32 editor.

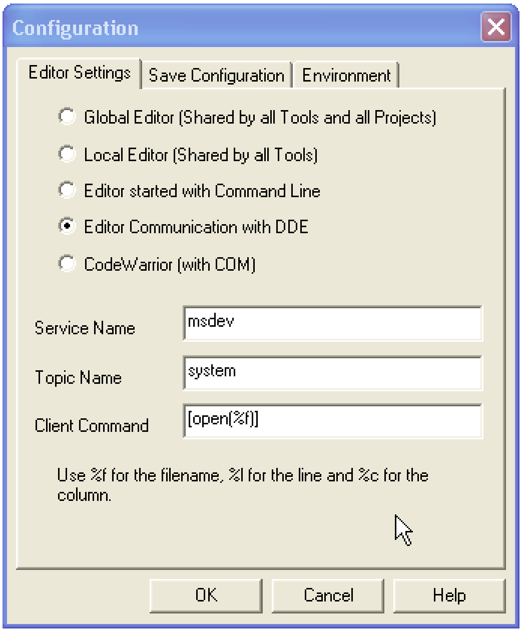
Listing 2.1 UltraEdit-32 configuration

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

Editor started with DDE

Enter the service, topic and client name to be used for a DDE (Dynamic Data Exchange) connection to the editor (Figure 2.7 on page 100). All entries can have modifiers for the filename and line number, as explained in the Modifiers on page 101 section.

Figure 2.7 DDE Editor configuration



○ Global Editor (Shared by all Tools and all Projects)
○ Local Editor (Shared by all Tools)
○ Editor started with Command Line
● Editor Communication with DDE
○ CodeWarrior (with COM)

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

Use %f for the filename, %l for the line and %c for the column.

For the Microsoft Developer Studio, use the settings in Listing 2.2 on page 100:

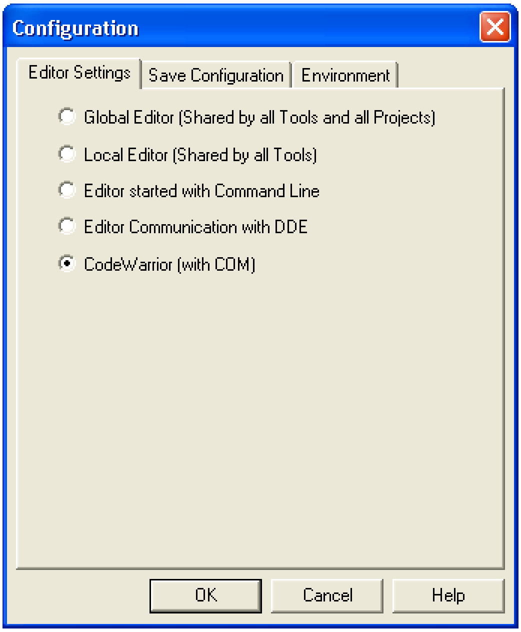
Listing 2.2 Microsoft Developer Studio configuration settings

```
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 2.8 on page 101).

Figure 2.8 COM Editor Configuration



○ Global Editor (Shared by all Tools and all Projects)

○ Local Editor (Shared by all Tools)

○ Editor started with Command Line

○ Editor Communication with DDE

● CodeWarrior (with COM)

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:*

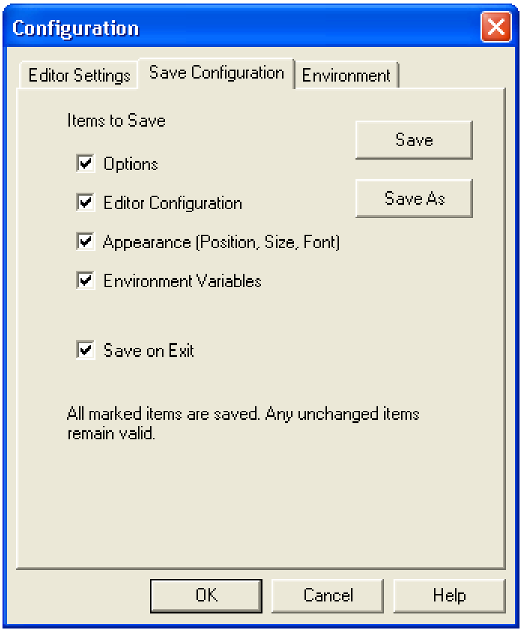
```
C:\WINAPPS\WINEDIT\Winedit.exe %f
```

NOTE *Please 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 2.9 on page 102).

Figure 2.9 Save Configuration dialog box



Items to Save

☑ Options

☑ Editor Configuration

☑ Appearance (Position, Size, Font)

☑ Environment Variables

☑ Save on Exit

All marked items are saved. Any unchanged items remain valid.

In the *Save Configuration* index, there are four check boxes where you can choose which items to save into a project file when the configuration is saved.

This dialog box has the following configurations:

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

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

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

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

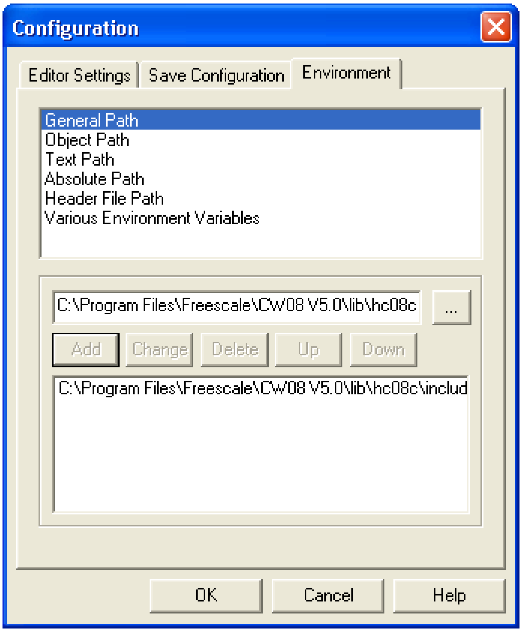
NOTE Almost all settings are stored in the project configuration file.
The only exceptions are:
- The recently used configuration list.
- All settings in the Save Configuration dialog box.

NOTE The configurations of the Assembler can, and in fact are intended to, coexist in the same file as the project configuration of other tools and the IDF. When an editor is configured by the shell, the assembler can read this content out of the project file, if present. The default project configuration filename is `project.ini`. The assembler automatically opens an existing `project.ini` in the current directory at startup. Also when using the -Prod: Specify project file at startup 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 2.10 on page 104).

Figure 2.10 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 2.5 on page 104):

Table 2.5 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:

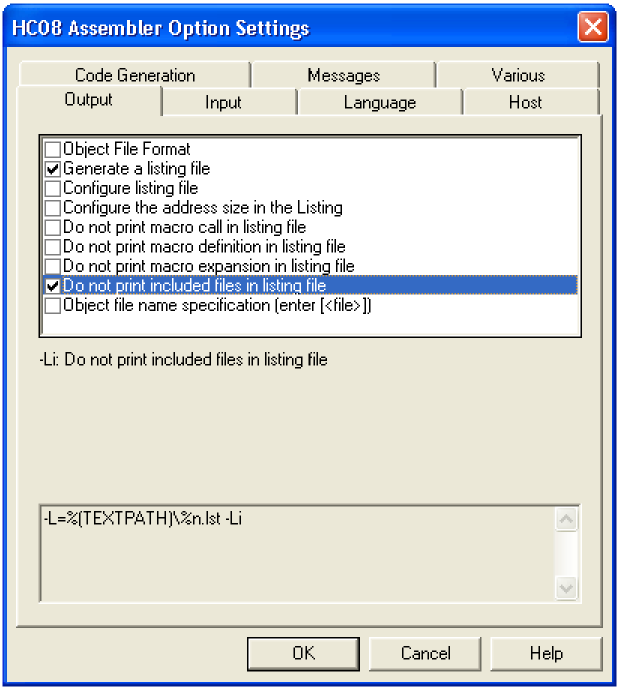
- *Add*: Adds a new line or entry
- *Change*: Changes a line or entry
- *Delete*: Deletes a line or entry
- *Up*: Moves a line or entry up
- *Down*: Moves a line or entry down

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 2.11 on page 105) to set or reset assembler options.

Figure 2.11 Option Settings dialog box



HC08 Assembler Option Settings

- □ Object File Format
- ☑ Generate a listing file
- □ Configure listing file
- □ Configure the address size in the Listing
- □ Do not print macro call in listing file
- □ Do not print macro definition in listing file
- □ Do not print macro expansion in listing file
- ☑ Do not print included files in listing file
- □ Object file name specification (enter [<file>])

-Li: Do not print included files in listing file

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

The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet (Table 2.6 on page 106):

Table 2.6 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++, ...)
Host	Lists options related to the host.
Code Generation	Lists options related to code generation (memory models, ...).
Messages	Lists options controlling the generation of error messages.
Various	Lists various additional options (options used for compatibility, ...).

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

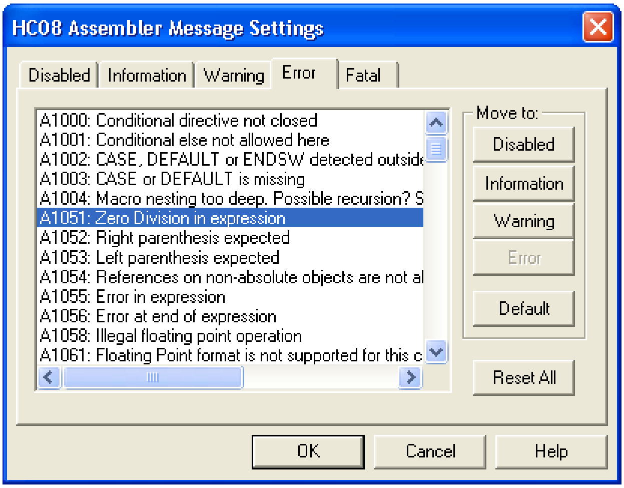
When the dialog box is opened and no option is selected, pressing the `F1` key or the *Help* button shows the help about this dialog box.

The available options are listed in the Assembler Options chapter.

Message settings dialog box

You can use the Message Settings (Table 2.7 on page 107) dialog box to map messages to a different message class.

Figure 2.12 Message Settings dialog box



A1000: Conditional directive not closed
A1001: Conditional else not allowed here
A1002: CASE, DEFAULT or ENDSW detected outside
A1003: CASE or DEFAULT is missing
A1004: Macro nesting too deep. Possible recursion? S
A1051: Zero Division in expression
A1052: Right parenthesis expected
A1053: Left parenthesis expected
A1054: References on non-absolute objects are not al
A1055: Error in expression
A1056: Error at end of expression
A1058: Illegal floating point operation
A1061: Floating Point format is not supported for this c

Some buttons in the dialog box may be disabled. For example, if an option cannot be moved to an information message, the '*Move to: Information*' button is disabled. The buttons in Table 2.7 on page 107 are available in the *Message Settings* dialog box:

Table 2.7 Message Settings options

Button	Description
Move to: Disabled	The selected messages are disabled; they will no longer be displayed.
Move to: Information	The selected messages are changed to information messages.
Move to: Warning	The selected messages are changed to warning messages.
Move to: Error	The selected messages are changed to error messages.
Move to: Default	The selected messages are changed to their default message types.
Reset All	Resets all messages to their default message types.
OK	Exits this dialog box and saves any changes.

Table 2.7 Message Settings options (*continued*)

Button	Description
Cancel	Exits this dialog box without accepting any changes.
Help	Displays online help about this dialog box.

A panel is available for each error message class and the content of the list box depends on the selected panel (Table 2.8 on page 108):

Table 2.8 Message classes

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

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

Changing the class associated with a message

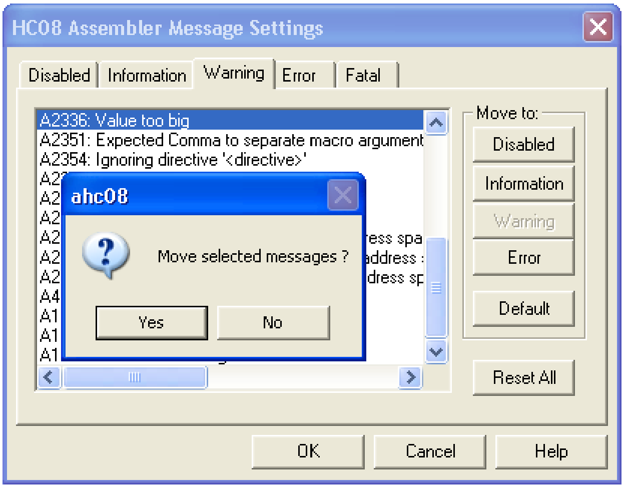
You can configure your own mapping of messages to the different classes. To do this, use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message, you have to select the message in the list box and then click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

Example

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

- Click the *Warning* sheet to display the list of all warning messages in the list box.
- Click on the *A2336: Value too big* string in the list box to select the message.
- Click *Error* to define this message as an error message. The *ahc08* dialog box appears. Press *Yes* to close the *ahc08* dialog box (Figure 2.13 on page 109).

Figure 2.13 HC08 Assembler Message Settings dialog box



A2336: Value too big
A2351: Expected Comma to separate macro argument
A2354: Ignoring directive '<directive>'

Move selected messages ?

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 *HC08 Assembler 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.

Press *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 the input file

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

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

Use the command line in the toolbar to assemble

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

Assembling a new file

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

Assembling a file which has already been assembled

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

Use the File > Assemble... entry

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

Use Drag and Drop

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

Message/Error feedback

After assembly, there are several ways to check where different errors or warnings have been detected. The default format of the error message is as on page 111. A typical error message is like the one in Listing 2.4 on page 111.

Listing 2.3 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 2.4 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 the assembler window

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

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

Use a user-defined editor

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

Line number can be specified on the command line

Editors like UltraEdit-32 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:

- Activate the assembler again.

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

3

Environment

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

There are three ways to specify an environment:

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

2) An optional ‘default.env’ file in the current directory. This file is supported for compatibility reasons with earlier versions. The name of this file may be specified using the ENVIRONMENT: Environment file specification on page 126 environment variable. Using the default.env file is not recommended.

3) Setting environment variables on system level (DOS level). This is also not recommended.

Various parameters of the Assembler may be set in an environment using so-called environment variables. The syntax is always the same (Listing 3.1 on page 115).

Listing 3.1 Syntax for setting environment variables

```
Parameter: KeyName=ParamDef
```

Listing 3.2 on page 115 is a typical example of setting an environment variable.

Listing 3.2 Setting the GENPATH environment variable

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

These parameters may be defined in several ways:

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

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

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

NOTE The environment may also be changed using the -Env: Set environment variable 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, ...).

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, ...), the current directory is the one specified by the launching tool.
- When a local project file is loaded, the current directory is set to the directory which contains the local project file. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for an assembly source file does not change the current directory.

To overwrite this behavior, the DEFAULTDIR: Default current directory on page 125 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 3.3 on page 117) in your environment settings.

Listing 3.3 Using a macro for setting environment variables

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

In the example in Listing 3.3 on page 117, `TEXTPATH` is expanded to '`C:\test\txt`', and `OBJPATH` is expanded to '`C:\test\obj`'.

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

In addition, the following special variables in Listing 3.4 on page 117 are allowed. Note that they are case-sensitive and always surrounded by `{}`. Also the variable content contains a directory separator '\' as well.

Listing 3.4 Special variables used with macros for setting environment variables

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

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

`{System}`
This is the path were your Windows O/S is installed, e.g., `'C:\WINNT\'`.

Global initialization file - 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 3.5 on page 118 shows two typical locations used for the mcutools.ini files.

Listing 3.5 Usual locations for the mcutools.ini files

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

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

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

Local configuration file (usually project.ini)

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

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

The current directory is always the directory that holds the configuration file. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the whole default.env file is also reloaded. When a configuration file is loaded or stored, the options located in the ASMOPTIONS: Default assembler options on page 123 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 3.6 on page 119.

Listing 3.6 Syntax used for setting path lists of environment variables

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

Listing 3.7 on page 119 is a typical example of setting an environment variable.

Listing 3.7 Setting the paths for the GEBNPATH environment variable

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

If a directory name is preceded by an asterisk (`*`), the programs recursively search that whole directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list. Listing 3.8 on page 119 shows the use of an asterisk (`*`) for recursively searching the entire C drive for a configuration file with a `\INSTALL\LIB` path.

Listing 3.8 Recursive search for a continuation line

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

NOTE Some DOS/UNIX environment variables (like `GENPATH`, `LIBPATH`, etc.) are used. For further details refer to Environment variables details on page 121.

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 3.9 on page 120):

Listing 3.9 Using multiple lines for an environment variable

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

Listing 3.9 on page 120 is the same as the alternate source code in Listing 3.10 on page 120.

Listing 3.10 Alternate form of using multiple lines

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

But this feature may be dangerous when used together with paths (Listing 3.11 on page 120).

Listing 3.11 A path is included by the line continuation character

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

In order to avoid such problems, we recommend that you use a semicolon `';'` at the end of a path if there is a backslash `'\'` at the end (Listing 3.12 on page 121 on page 121).

Listing 3.12 Recommended style whenever a backlash 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 3.1 on page 121).

Table 3.1 Topics used for describing environment variables

Topic	Description
Tools	Lists tools which are using this variable.
Synonym (where one exists)	A synonym exists for some environment variables. These synonyms may be used for older releases of the Assembler but they are deprecated and they will be removed in the future. A synonym has lower precedence than the environment variable.
Syntax	Specifies the syntax of the option in an EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default (if one exists)	Shows the default setting for the variable if one exists.
Description	Provides a detailed description of the option and its usage.
Example	Gives an example of usage and effects of the variable where possible. An example shows an entry in the default.env for the PC or in the .hidefaults for UNIX.
See also (if needed)	Names related sections.

ABSPATH: Absolute file path

Tools

Compiler, Assembler, Linker, Decoder, or Debugger

Syntax

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

Arguments

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

Description

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

Example

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

ASMOPTIONS: Default assembler options

Tools

Assembler

Syntax

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

Arguments

`<option>`: Assembler command-line option

Description

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

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

Example

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

See also

Assembler Options 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

Environment variables:

- USERNAME: User Name in object file on page 136
- INCLUDETIME: Creation time in the object file on page 131

DEFAULTDIR: Default current directory

Tools

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

Syntax

```
DEFAULTDIR=<directory>
```

Arguments

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

Description

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

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

Example

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

See also

“Current directory” on page 116

“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 3.13 on page 127 lists all errors into the `MyErrors.err` file in the current directory.

Listing 3.13 Naming an error file

```
ERRORFILE=MyErrors.err
```

Listing 3.14 on page 128 lists all errors into the `errors` file in the `\tmp` directory.

Listing 3.14 Naming an error file in a specific directory

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

Listing 3.15 on page 128 lists all errors into a file with the same name as the source file, but with extension `*.err`, into the same directory as the source file, e.g., if we compile a file `\sources\test.c`, an error list file `\sources\test.err` will be generated.

Listing 3.15 Naming an error file as source filename

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

For a `test.c` source file, a `\dir1\test.err` error list file will be generated (Listing 3.16 on page 128).

Listing 3.16 Naming an error file as source filename in a specific directory

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

For a `\dir1\dir2\test.c` source file, a `\dir1\dir2\errors.txt` error list file will be generated (Listing 3.17 on page 128).

Listing 3.17 Naming an error file as a source filename with full path

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

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

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

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

Another example (Listing 3.18 on page 129) shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called `EDOUT`:

Listing 3.18 Configuring error feedback with WinEdit

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

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

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

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

GENPATH: Search path for input file

Tools

Compiler, Assembler, Linker, Decoder, or Debugger

Synonym

`HIPATH`

Syntax

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

Arguments

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

Description

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

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

Example

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

INCLUDETIME: Creation time in the object file

Tools

Compiler, Assembler, Linker, or Librarian

Syntax

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

Arguments

`ON`: Include time information into the object file.

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

Default

```
ON
-
```

Description

Normally each object file created contains a time stamp indicating the creation time and data as strings. So whenever a new file is created by one of the tools, the new file gets a new time stamp entry.

This behavior may be undesired if for SQA reasons a binary file compare has to be performed. Even if the information in two object files is the same, the files do not match exactly because the time stamps are not the same. To avoid such problems this variable may be set to `OFF`. In this case the time stamp strings in the object file for date and time are “`none`” in the object file.

The time stamp may be retrieved from the object files using the Decoder.

Example

```
INCLUDETIME=OFF
```

See also

Environment variables:

- COPYRIGHT: Copyright entry in object file on page 124
- USERNAME: User Name in object file on page 136

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 on page 116 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

Environment variables:

- COPYRIGHT: Copyright entry in object file on page 124
- INCLUDETIME: Creation time in the object file on page 131

4

Files

This chapter covers these topics:

- Input files on page 137
- Output files on page 137
- Output files on page 137

Input files

Input files to the Assembler:

- Source files on page 137
- Object files on page 138

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 on page 138
- Absolute files on page 138
- S-Record Files on page 138

- Listing files on page 139
- Debug listing files on page 139
- Error listing file on page 139

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)

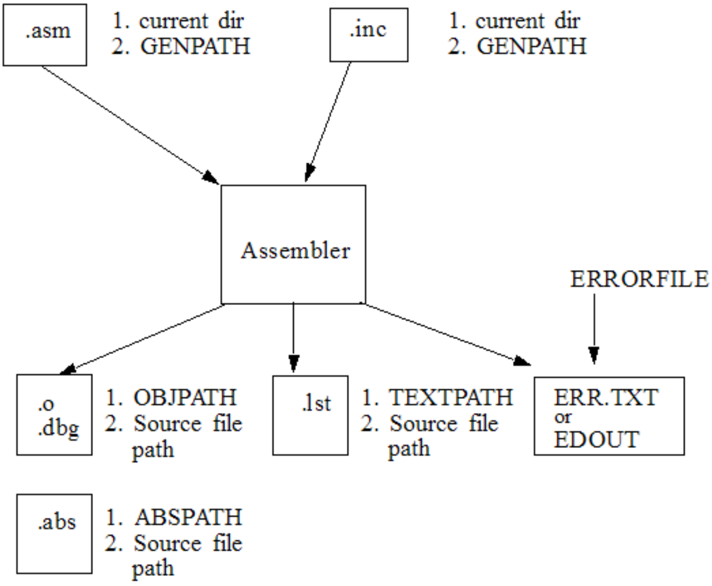
If ERRORFILE is set, the Assembler creates a message file named as specified in this environment variable.

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

File Processing

Figure 4.1 on page 140 shows the priority levels for the various files used by the Assembler.

Figure 4.1 Files used with the Assembler



5

Assembler Options

Types of assembler options

The Assembler offers a number of assembler options that you can use to control the Assembler’s operation. Options are composed of a dash/minus `(-)` followed by one or more letters or digits. Anything not starting with a dash/minus is supposed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the ASMOPTIONS: Default assembler options (Table 5.1 on page 143) environment variable. Typically, each Assembler option is specified only once per assembling session.

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

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

Table 5.1 ASMOPTIONS environment variable

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

Assembler options (Table 5.2 on page 144) are grouped by:

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

Table 5.2 Assembler option categories

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

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

Each option has also a scope (Table 5.3 on page 144)

Table 5.3 Scopes for assembler options

Scope	Description
Application	This option has to be set for all files (assembly units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.
Assembly Unit	This option can be set for each assembling unit for an application differently. Mixing objects in an application is possible.
None	The scope option is not related to a specific code part. A typical example are options for the message management.

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

Assembler Option details

The remainder of this section is devoted to describing each of the assembler options available for the Assembler. The options are listed in alphabetical order and each is divided into several sections (Table 5.4 on page 145).

Table 5.4 Assembler option details

Topic	Description
Group	Output, Input, Language, Host, Code Generation, Messages, or Various.
Scope	Application, Assembly Unit, Function, or None.
Syntax	Specifies the syntax of the option in an EBNF format.
Arguments	Describes and lists optional and required arguments for the option.
Default	Shows the default setting for the option.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and effects of the option where possible. Assembler settings, source code and/or Linker PRM files are displayed where applicable. The examples shows an entry in the `default.env` for the PC or in the `.hidefaults` for UNIX.
See also (if needed)	Names related options.

Using special modifiers

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

The following modifiers are supported (Table 5.5 on page 145)

Table 5.5 Special modifiers for assembler options

Modifier	Description
%p	Path including file separator
%N	Filename in strict 8.3 format

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

Modifier	Description
%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 5.1 on page 146.

Listing 5.1 Example filename and path used for the following examples

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

Using the `%p` modifier as in Listing 5.2 on page 146 displays the path with a file separator but without the filename.

Listing 5.2 %p gives the path only with the final file separator

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

Using the `%N` modifier only displays the filename in 8.3 format but without the file extension (Listing 5.3 on page 147).

Listing 5.3 %N results in the filename in 8.3 format (only the first 8 characters)

```
TheWhole
```

The `%n` modifier returns the entire filename but with no file extension (Listing 5.4 on page 147.

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

```
TheWholeThing
```

Using `%E` as a modifier returns the first three characters in the file extension (Listing 5.5 on page 147).

Listing 5.5 %E gives the file extension in 8.3 format (only the first 3 characters)

```
myE
```

If you want the entire file extension, use the `%e` modifier (Listing 5.6 on page 147).

Listing 5.6 %e is used for returning the whole extension

```
myExt
```

The `%f` modifier returns the path and the filename but without the file extension (Listing 5.7 on page 147).

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

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

The path in Listing 5.1 on page 146 contains a space, therefore using `%”` or `%’` is recommended
(Listing 5.8 on page 148 or Listing 5.9 on page 148).

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

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

Listing 5.9 Use %’%f%’ where there is a space in its path, filename, or extension

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

Using `%(envVariable)` an environment variable may be used. A file separator following `%(envVariable)` is ignored if the environment variable is empty or does not exist. If `TEXTPATH` is set as in Listing 5.10 on page 148, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.11 on page 148.

Listing 5.10 Example for setting TEXTPATH

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

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

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

However, if `TEXTPATH` does not exist or is empty, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.12 on page 148).

Listing 5.12 $(TEXTPATH)\myfile.txt where TEXTPATH does not exist

```
myfile.txt
```

It is also possible to display the percent sign by using %%. %e%% allows the expression of a percent sign after the extension as in Listing 5.13 on page 149.

Listing 5.13 %% allows a percent sign to be expressed

```
myExt%
```

List of Assembler options

The following table lists each command line option you can use with the Assembler (Table 5.6 on page 149).

Table 5.6 Assembler options

Assembler option
-Ci: Switch case sensitivity on label names OFF on page 152
-CMacAngBrack: Angle brackets for grouping Macro Arguments on page 154
-CMacBrackets: Square brackets for macro arguments grouping on page 155
-Compat: Compatibility modes on page 156
-CS08/-C08/-CRS08: Derivative family on page 159
-D: Define Label on page 161
-Env: Set environment variable on page 163
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output-file format on page 164
-H: Short Help on page 166
-I: Include file path on page 167
-L: Generate a listing file on page 168
-Lasmc: Configure listing file on page 171
-Lasms: Configure the address size in the listing file on page 173
-Lc: No Macro call in listing file on page 175
-Ld: No macro definition in listing file on page 178

Table 5.6 Assembler options (*continued*)

Assembler option
-Le: No Macro expansion in listing file on page 181
-Li: No included file in listing file on page 184
-Lic: License information on page 186
-LicA: License information about every feature in directory on page 187
-LicBorrow: Borrow license feature on page 188
-LicWait: Wait until floating license is available from floating License Server on page 190
-Ll: Show label statistics on page 191
-M (-Ms, -Mt): Memory model on page 193
-MacroNest: Configure maximum macro nesting on page 195
-MCUasm: Switch compatibility with MCUasm ON on page 196
-N: Display notify box on page 197
-NoBeep: No beep in case of an error on page 198
-NoDebugInfo: No debug information for ELF/DWARF files on page 199
-NoEnv: Do not use environment on page 200
-ObjN: Object filename specification on page 201
-Prod: Specify project file at startup on page 203
-Struct: Support for structured types on page 204
-V: Prints the Assembler version on page 205
-View: Application standard occurrence on page 206
-W1: No information messages on page 208
-W2: No information and warning messages on page 209
-WErrFile: Create "err.log" error file on page 210
-Wmsg8x3: Cut filenames in Microsoft format to 8.3 on page 211
-WmsgCE: RGB color for error messages on page 212
-WmsgCF: RGB color for fatal messages on page 213

Table 5.6 Assembler options (*continued*)

Assembler option
-WmsgCI: RGB color for information messages on page 214
-WmsgCU: RGB color for user messages on page 215
-WmsgCW: RGB color for warning messages on page 216
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 217
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
-WmsgFob: Message format for batch mode on page 221
-WmsgFoi: Message format for interactive mode on page 223
-WmsgFonf: Message format for no file information on page 225
-WmsgFonp: Message format for no position information on page 227
-WmsgNe: Number of error messages on page 229
-WmsgNi: Number of Information messages on page 230
-WmsgNu: Disable user messages on page 231
-WmsgNw: Number of Warning messages on page 233
-WmsgSd: Setting a message to disable on page 234
-WmsgSe: Setting a message to Error on page 235
-WmsgSi: Setting a message to Information on page 236
-WmsgSw: Setting a Message to Warning on page 237
-WOutFile: Create error listing file on page 238
-WStdout: Write to standard output on page 239

Detailed listing of all assembler options

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

-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 5.14 on page 153.

Listing 5.14 Example assembly source code

```
       ORG $200
entry: NOP
       BRA Entry
```

The instruction 'BRA Entry' branches on the 'entry' label. The default setting for case sensitivity is ON, which means that the Assembler interprets the labels 'Entry' and 'entry' as two distinct labels.

See also

-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output-file format on page 164 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. New code should use the `[? ?]` syntax instead.

See also

Macro argument grouping

-CMacBrackets: Square brackets for macro arguments grouping on page 155 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 control whether the `[? ?]` syntax for macro invocation argument grouping is available. When it is disabled, the Assembler does not recognize the special meaning for `[?` in the macro invocation context.

See also

Macro argument grouping

-CMacAngBrack: Angle brackets for grouping Macro Arguments on page 154 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 5.15 on page 157 demonstrates that when `-Compat=c`, comments can start with a `*`.

Listing 5.15 Comments starting with an asterisk (*)

```
NOP   * Anything following an asterisk is a comment.
```

When the `-Compat=c` assembler option is used, the first `DC.B` directive in Listing 5.16 on page 158 has `"+ 1 , 1"` as a comment . A warning is issued because the "comment" does not start with a `";"` or a `"*"`. With `-Compat=c`, this code generates a warning and three bytes with constant values 1, 2, and 1. Without it, this code generates four 8-bit constants of 2, 1, 2, and 1.

Listing 5.16 Implicit comment start after a space

```
DC.B 1 + 1 , 1
DC.B 1+1,1
```

- `s`: Symbol prefixes

  With this suboption, some compatibility prefixes for symbols are supported. With this option, the Assembler accepts “`pgz:`” and “`byte:`” prefixed for symbols in `XDEFs` and `XREFs`. They correspond to `XREF.B` or `XDEF.B` with the same symbols without the prefix.

- `f`: Ignore `FF` character at line start

  With this suboption, an otherwise improper character recognized from 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 3 different HC08 derived cores. The HC08 itself (`-C08`), the enhanced HCS08 (`-CS08`) and the reduced 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 5.7 Table of new instructions or addressing modes for the HCS08

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

-D: Define Label

Group

Input

Scope

Assembly Unit

Syntax

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

Arguments

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

Default

`0` for `Value`.

Description

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

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

Example

Conditional inclusion of a copyright notice. See Listing 5.17 on page 161 and Listing 5.18 on page 162.

Listing 5.17 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 5.18 on page 162 is generated:

Listing 5.18 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
```

-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” on page 121

-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 5.19 on page 166 is a portion of the list produced by the `-H` option:

Listing 5.19 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” on page 145).

Default

No generated listing file

Description

Switches on the generation of the listing file. If `dest` is not specified, the listing file will have the same name as the source file, but with extension `*.lst`. The listing file contains macro definition, invocation, and expansion lines as well as expanded include files.

Example

```
ASMOPTIONS=-L
```

In the following example of assembly code (Listing 5.20 on page 169), the cpChar macro accepts two parameters. The macro copies the value of the first parameter to the second one.

When the `-L` option is specified, the portion of assembly source code in Listing 5.20 on page 169, together with the code from an include file (Listing 5.21 on page 169) generates the output listing in Listing 5.22 on page 169.

Listing 5.20 Example assembly source code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

Listing 5.21 Example source code from an include file

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

Listing 5.22   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 on page 171
- -Lasms: Configure the address size in the listing file on page 173
- -Lc: No Macro call in listing file on page 175
- -Ld: No macro definition in listing file on page 178
- -Le: No Macro expansion in listing file on page 181
- -Li: No included file in listing file on page 184

-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 on page 175, -Ld: No macro definition in listing file on page 178, -Le: No Macro expansion in listing file on page 181, and -Li: No included file in listing file on page 184.

Example

For the following assembly source code, the Assembler generates the default-configured output listing (Listing 5.23 on page 172):

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

Listing 5.23 Example assembler output listing

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

In order to get this output without the source file line numbers and other irrelevant parts for this simple `DC.B` example, the following option is added: `"-Lasmc=ramki"`. This generates the output listing in Listing 5.24 on page 172:

Listing 5.24 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 on page 168
- -Lc: No Macro call in listing file on page 175
- -Ld: No macro definition in listing file on page 178
- -Le: No Macro expansion in listing file on page 181
- -Li: No included file in listing file on page 184
- -Lasms: Configure the address size in the listing file on page 173

-Lasms: Configure the address size in the listing file

Group

Output

Scope

Assembly unit

Syntax

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

Arguments

1 - The address size is xx

2 - The address size is xxxx

3 - The address size is xxxxxx

4 - The address size is xxxxxxxx

Default

```
-Lasms3
```

Description

The default-configured listing file shows a lot of information. With this option, the size of the address column can be reduced to the size of interest. To configure which columns are printed, see the -Lasmc: Configure listing file on page 171 option. To configure which lines to print, see the -Lc: No Macro call in listing file on page 175, -Ld: No macro definition in listing file on page 178, -Le: No Macro expansion in listing file on page 181, and -Li: No included file in listing file on page 184 assembler options.

Example

For the following instruction:

```
NOP
```

the Assembler generates this default-configured output listing (Listing 5.25 on page 174):

Listing 5.25 Example assembler output listing

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1   000000 XX                 NOP
```

In order to change the size of the address column the following option is added: `"-Lasms1"`. This changes the address size to two digits.

Listing 5.26 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 on page 171
- -L: Generate a listing file on page 168
- -Lc: No Macro call in listing file on page 175
- -Ld: No macro definition in listing file on page 178
- -Le: No Macro expansion in listing file on page 181
- -Li: No included file in listing file on page 184

-Lc: No Macro call in listing file

Group

Output

Scope

Assembly unit

Syntax

```
-Lc
```

Arguments

none

Default

none

Description

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

Example

```
ASMOPTIONS=-Lc
```

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

When the -Lc option is specified, the following portion of assembly source code in Listing 5.27 on page 175:

Listing 5.27 Example assembly source code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
```

```
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

along with additional source code (Listing 5.28 on page 176) from the `macro.inc` include file generates the following output in the assembly listing file (Listing 5.29 on page 176):

Listing 5.28 Example source code from the macro.inc file

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

Listing 5.29 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 on page 168
- -Ld: No macro definition in listing file on page 178
- -Le: No Macro expansion in listing file on page 181
- -Li: No included file in listing file on page 184

-Ld: No macro definition in listing file

Group

Output

Scope

Assembly unit

Syntax

`-Ld`

Arguments

None

Default

None

Description

Instructs the Assembler to generate a listing file but not including any macro definitions. The listing file contains macro invocation and expansion lines as well as expanded include files.

Example

`ASMOPTIONS=-Ld`

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

When the `-Ld` option is specified, the assembly source code in Listing 5.30 on page 178 along with additional source code (Listing 5.31 on page 179) from the `macro.inc` file generates an assembler output listing (Listing 5.32 on page 179) file:

Listing 5.30 Example assembly source code

```
         XDEF  Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
```

```
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

Listing 5.31 Example source code from an include file

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

Listing 5.32 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 on page 168
- -Lc: No Macro call in listing file on page 175
- -Le: No Macro expansion in listing file on page 181

- -Li: No included file in listing file on page 184

-Le: No Macro expansion in listing file

Group

Output

Scope

Assembly unit

Syntax

```
-Le
```

Arguments

None

Default

None

Description

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

Example

```
ASMOPTIONS=-Le
```

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

When the `-Le` option is specified, the assembly code in Listing 5.33 on page 181 along with additional source code (Listing 5.34 on page 182) from the `macro.inc` file generates an assembly output listing file (Listing 5.35 on page 182):

Listing 5.33 Example assembly source code

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
```

```
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

Listing 5.34 Example source code from an included file

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

Listing 5.35 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 on page 168

-Lc: No Macro call in listing file on page 175

-Ld: No macro definition in listing file on page 178-Li: No included file in listing file on page 184

-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 5.36 on page 184 along with additional source code (Listing 5.37 on page 185) from the `macro.inc` file generates the following output in the assembly listing file:

Listing 5.36 Example assembly source code

```
         XDEF  Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
```

```
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

Listing 5.37 Example source code in an include file

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

Listing 5.38 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 on page 168
- -Lc: No Macro call in listing file on page 175
- -Ld: No macro definition in listing file on page 178
- -Le: No Macro expansion in listing file on page 181

-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 on page 187
- -LicBorrow: Borrow license feature on page 188
- -LicWait: Wait until floating license is available from floating License Server on page 190

-LicA: License information about every feature in directory

Group

Various

Scope

None

Syntax

```
-LicA
```

Arguments

None

Default

None

Description

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

Example

```
ASMOPTIONS=-LicA
```

See also

Assembler options:

- -Lic: License information on page 186
- -LicBorrow: Borrow license feature on page 188
- -LicWait: Wait until floating license is available from floating License Server on page 190

-LicBorrow: Borrow license feature

Group

Host

Scope

None

Syntax

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

Arguments

`<feature>`: the feature name to be borrowed (e.g., `HI100100`).
`<version>`: optional version of the feature to be borrowed (e.g., `3.000`).
`<date>`: date with optional time until when the feature shall be borrowed (e.g., `15-Mar-2005:18:35`).

Default

None

Defines

None

Pragmas

None

Description

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

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

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

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

Example

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

See also

Assembler options:

- -Lic: License information on page 186
- -LicA: License information about every feature in directory on page 187
- -LicWait: Wait until floating license is available from floating License Server on page 190

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

Group

Host

Scope

None

Syntax

```
-LicWait
```

Arguments

None

Default

None

Description

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

Example

```
ASMOPTIONS=-LicWait
```

See also

Assembler options:

- -Lic: License information on page 186
- -LicA: License information about every feature in directory on page 187
- -LicBorrow: Borrow license feature on page 188

-Ll: Show label statistics

Group

OUTPUT

Scope

Assembly unit

Syntax

```
"-Ll"
```

Arguments

None

Default

No label statistics

Description

Appends a section to the listing file that shows how much space would be gained if a certain label were in the short (or tiny) memory area. The statistics take into consideration the relocatable symbols only (labels defined by an EQU directive are not taken into consideration). This option has no effect if no listing file is generated (i.e. -L is not active too). This is an RS08 specific option, and is not supported for any other HC08 derivative.

Example:

```
  XREF label_a, label_b
MY_CODE_SECTION: SECTION
label_c:
  LDA label_a
  STA label_b
  INCA
  ADD label_b
  LDA label_c
```

Assembling the code above (with `-L -Ll`) results in the following listing file:

```
Freescale HC08-Assembler
(c) Copyright Freescale 1987-2006

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                        XREF label_a, label_b
    2    2                      MY_CODE_SECTION: SECTION
    3    3                      label_c:
    4    4   000000 B6 xx         LDA label_a
    5    5   000002 B7 xx         STA label_b
    6    6   000004 4C            INCA
    7    7   000005 BB xx         ADD label_b
    8    8   000007 B6 xx         LDA label_c
    9    9

Freescale Assembler
 Ind.  Name             tiny       short
 ----  ----             ----       -----
    1  label_a                 1          1
    2  label_b                 2          1
    3  label_c                 1          1
```

The table at the end of the listing file shows that:

1) If label_a one were either in the tiny or short memory area, one byte would be gained in terms of code size (since the LDA at line 4 would use the short addressing mode in either of the two cases).

2) If label_b were in the tiny memory area, two bytes would be gained since the STA at line 5 would use the short addressing mode and the ADD at line 7 would use the tiny addressing mode.

3) If label_c were in the short (or tiny) memory area one byte would be gained since the LDA at line 8 would use the short addressing mode.

-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
```

-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 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 5.39 on page 205):

Listing 5.39 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 (e.g., Assembler, Linker, Compiler, ...) is started with a normal window if no arguments are given. If the application is started with arguments (e.g., from the Maker to assemble, compile, or link a file), then the application is running minimized to allow for batch processing. However, the application's window behavior may be specified with the View option.

Using `-ViewWindow`, the application is visible with its normal window. Using `-ViewMin` the application is visible iconified (in the task bar). Using `-ViewMax`, the application is visible maximized (filling the whole screen). Using `-ViewHidden`, the application processes arguments (e.g., files to be compiled or linked) completely invisible in the background (no window or icon visible in the task bar). However, for example, if you are using the -N: Display notify box on page 197 assembler option, a dialog box is still possible.

Example

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

-W1: No information messages

Group

Messages

Scope

Assembly Unit

Syntax

```
-W1
```

Arguments

None

Default

None

Description

Inhibits the Assembler's printing INFORMATION messages. Only WARNING and ERROR messages are written to the error listing file and to the assembler window.

Example

```
ASMOPTIONS=-W1
```

-W2: No information and warning messages

Group

Messages

Scope

Assembly Unit

Syntax

```
-W2
```

Arguments

None

Default

None

Description

Suppresses all messages of INFORMATION or WARNING types. Only ERROR messages are written to the error listing file and to the assembler window.

Example

```
ASMOPTIONS=-W2
```

-WErrFile: Create "err.log" error file

Group

Messages

Scope

Assembly Unit

Syntax

```
-WErrFile(On|Off)
Arguments
```

None

Default

`An err.log` file is created or deleted.

Description

The error feedback from the Assembler to called tools is now done with a return code. In 16-bit Windows environments this was not possible. So in case of an error, an “err.log” file with the numbers of written errors was used to signal any errors. To indicate no errors, the “err.log”file would be deleted. Using UNIX or WIN32, a return code is now available. Therefore, this file is no longer needed when only UNIX or WIN32 applications are involved. To use a 16-bit Maker with this tool, an error file must be created in order to signal any error.

Example

- `-WErrFileOn`

  `err.log` is created or deleted when the application is finished.

- `-WErrFileOff`

  existing `err.log` is not modified.

See also

-WStdout: Write to standard output on page 239

-WOutFile: Create error listing file on page 238

-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 Win95, WinNT, or a newer Windows O/S, longer file names are possible. With this option the filename in the Microsoft message is truncated to the 8.3 format.

Example

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

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

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

See also

- -WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode on page 217
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
- -WmsgFoi: Message format for interactive mode on page 223
- -WmsgFob: Message format for batch mode on page 221 Option

- --WmsgFonp: Message format for no position information on page 227

-WmsgCE: RGB color for error messages

Group

Messages

Scope

Compilation Unit

Syntax

```
-WmsgCE<RGB>
```

Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

Default

```
-WmsgCE16711680 (rFF g00 b00, red)
```

Description

It is possible to change the error message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

Example

`-WmsgCE255` changes the error messages to blue.

-WmsgCF: RGB color for fatal messages

Group

Messages

Scope

Compilation Unit

Syntax

```
-WmsgCF<RGB>
```

Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

Default

```
-WmsgCF8388608 (r80 g00 b00, dark red)
```

Description

It is possible to change the fatal message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

Example

`-WmsgCF255` changes the fatal messages to blue.

-WmsgCI: RGB color for information messages

Group

Messages

Scope

Compilation Unit

Syntax

```
-WmsgCI<RGB>
```

Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

Default

```
-WmsgCI32768 (r00 g80 b00, green)
```

Description

It is possible to change the information message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

Example

`-WmsgCI255` changes the information messages to blue.

-WmsgCU: RGB color for user messages

Group

Messages

Scope

Compilation Unit

Syntax

```
-WmsgCU<RGB>
```

Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

Default

```
-WmsgCU0 (r00 g00 b00, black)
```

Description

It is possible to change the user message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

Example

`-WmsgCU255` changes the user messages to blue.

-WmsgCW: RGB color for warning messages

Group

Messages

Scope

Compilation Unit

Syntax

```
-WmsgCW<RGB>
```

Arguments

`<RGB>`: 24-bit RGB (red green blue) value.

Default

```
-WmsgCW255 (r00 g00 bFF, blue)
```

Description

It is possible to change the warning message color with this option. The value to be specified has to be an RGB (Red-Green-Blue) value and has to be specified in decimal.

Example

`-WmsgCW0` changes the warning messages to black.

-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFb[v|m]
```

Arguments

`v`: Verbose format.

`m`: Microsoft format.

Default

```
-WmsgFbm
```

Description

The Assembler can be started with additional arguments (e.g., files to be assembled together with assembler options). If the Assembler has been started with arguments (e.g., from the *Make tool*), the Assembler works in the batch mode. That is, no assembler window is visible and the Assembler terminates after job completion.

If the Assembler is in batch mode, the Assembler messages are written to a file and are not visible on the screen. This file only contains assembler messages (see examples below).

The Assembler uses a *Microsoft* message format as the default to write the assembler messages (errors, warnings, or information messages) if the Assembler is in the batch mode.

With this option, the default format may be changed from the *Microsoft* format (with only line information) to a more verbose error format with line, column, and source information.

Example

Assume that the assembly source code in Listing 5.40 on page 218 is to be assembled in the batch mode.

Listing 5.40 Example assembly source code

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

The Assembler generates the error output (Listing 5.41 on page 218) in the assembler window if it is running in batch mode:

Listing 5.41 Example error listing in the Microsoft (default) format for batch mode

```
X:\TW2.ASM(12):ERROR: Conditional else not allowed here.
```

If the format is set to verbose, more information is stored in the file:

Listing 5.42 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 on page 219

-WmsgFob: Message format for batch mode on page 221

-WmsgFoi: Message format for interactive mode on page 223

-WmsgFonf: Message format for no file information on page 225

-WmsgFonp: Message format for no position information on page 227

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

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgFi[v|m]
```

Arguments

`v`: Verbose format.

`m`: Microsoft format.

Default

```
-WmsgFiv
```

Description

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

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

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

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

Example

If the Assembler is running in interactive mode, the default error output is shown in the assembler window as in Listing 5.44 on page 220.

Listing 5.43 Example error listing in the default mode for interactive mode

```
>> in "X:\TWE.ASM", line 12, col 0, pos 215
         endif
         endif
^
ERROR A1001: Conditional else not allowed here
```

Setting the format to Microsoft, less information is displayed:

Listing 5.44 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 on page 217
- -WmsgFob: Message format for batch mode on page 221
- -WmsgFoi: Message format for interactive mode on page 223
- -WmsgFonf: Message format for no file information on page 225
- -WmsgFonp: Message format for no position information on page 227

-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 5.45 on page 221 are supported (assumed that the source file is `x:\Freescale\sourcefile.asmx).`

Listing 5.45 Supported formats for messages in the batch node

```
Format  Description  Example

----------------------------------------------------
%s     Source Extract
%p     Path                 x:\Freescale\
%f     Path and name        x:\Freescale\sourcefile
%n     Filename             sourcefile
%e     Extension           .asmx
%N     File (8 chars)       sourcefi
%E     Extension (3 chars)  .asm
%l     Line                 3
%c     Column               47
%o     Pos                  1234
%K     Uppercase kind       ERROR
%k     Lowercase kind       error
```

```
%d     Number                   A1051
%m     Message                  text
%%     Percent                  %
\n     New line
```

Example

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

produces a message displayed in `Listing 5.46 on page 222` using the format in Listing 5.45 on page 221. The options are set for producing the path of a file with its filename , extension, and line

Listing 5.46 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 on page 217
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
- -WmsgFoi: Message format for interactive mode on page 223
- -WmsgFonf: Message format for no file information on page 225
- -WmsgFonp: Message format for no position information on page 227

-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 5.47 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 5.48 on page 224):

Listing 5.48 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 on page 217
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
- -WmsgFob: Message format for batch mode on page 221
- -WmsgFonf: Message format for no file information on page 225
- -WmsgFonp: Message format for no position information on page 227

-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:

Listing 5.49

```
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 on page 217
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
- -WmsgFob: Message format for batch mode on page 221
- -WmsgFoi: Message format for interactive mode on page 223
- -WmsgFonp: Message format for no position information on page 227

-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 5.50 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 on page 217
- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode on page 219
- -WmsgFob: Message format for batch mode on page 221
- -WmsgFoi: Message format for interactive mode on page 223
- -WmsgFonf: Message format for no file information on page 225

-WmsgNe: Number of error messages

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgNe<number>
```

Arguments

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

Default

```
50
```

Description

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

Example

```
ASMOPTIONS=-WmsgNe2
```

The Assembler stops assembling after two error messages.

See also

Assembler options:

- -WmsgNi: Number of Information messages on page 230
- -WmsgNw: Number of Warning messages on page 233

-WmsgNi: Number of Information messages

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgNi<number>
```

Arguments

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

Default

```
50
```

Description

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

Example

```
ASMOPTIONS=-WmsgNi10
```

Only ten information messages are logged.

See also

Assembler options:

- -WmsgNe: Number of error messages on page 229
- -WmsgNw: Number of Warning messages on page 233

-WmsgNu: Disable user messages

Group

Messages

Scope

None

Syntax

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

Arguments

`a`: Disable messages about include files

`b`: Disable messages about reading files

`c`: Disable messages about generated files

`d`: Disable messages about processing statistics

`e`: Disable informal messages

Default

None

Description

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

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

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

NOTE Depending on the application, not all suboptions may make sense. In 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 on page 229
- -WmsgNi: Number of Information messages on page 230

-WmsgSd: Setting a message to disable

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgSd<number>
```

Arguments

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

Default

None

Description

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

Example

```
-WmsgSd1801
```

See also

Assembler options:

- -WmsgSe: Setting a message to Error on page 235
- -WmsgSi: Setting a message to Information on page 236
- -WmsgSw: Setting a Message to Warning on page 237

-WmsgSe: Setting a message to Error

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgSe<number>
```

Arguments

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

Default

None

Description

Allows changing a message to an error message.

Example

```
-WmsgSe1853
```

See also

Assembler options:

- -WmsgSd: Setting a message to disable on page 234
- -WmsgSi: Setting a message to Information on page 236
- -WmsgSw: Setting a Message to Warning on page 237

-WmsgSi: Setting a message to Information

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgSi<number>
```

Arguments

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

Default

None

Description

With this option a message can be set to an information message.

Example

```
-WmsgSi1853
```

See also

Assembler options:

- -WmsgSd: Setting a message to disable on page 234
- -WmsgSe: Setting a message to Error on page 235
- -WmsgSw: Setting a Message to Warning on page 237

-WmsgSw: Setting a Message to Warning

Group

Messages

Scope

Assembly Unit

Syntax

```
-WmsgSw<number>
```

Arguments

`<number>`: Error number to be a warning, e.g., `2901`

Default

None

Description

With this option a message can be set to a warning message.

Example

```
-WmsgSw2901
```

See also

Assembler options:

- -WmsgSd: Setting a message to disable on page 234
- -WmsgSe: Setting a message to Error on page 235
- -WmsgSi: Setting a message to Information on page 236

-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 on page 210
- -WStdout: Write to standard output on page 239

-WStdout: Write to standard output

Group

Messages

Scope

Assembly Unit

Syntax

`-WStdout(On|Off)`

Arguments

None

Default

output is written to `stdout`

Description

With Windows applications, the usual standard streams are available. But text written into them does not appear anywhere unless explicitly requested by the calling application. With this option is can be controlled if the text to error file should also be written into `stdout`.

Example

`-WStdoutOn`

All messages are written to `stdout`.

`-WErrFileOff`

Nothing is written to `stdout`.

See also

Assembler options:

- -WErrFile: Create "err.log" error file on page 210
- -WOutFile: Create error listing file on page 238

6

Sections

Sections are portions of code or data that cannot be split into smaller elements. Each section has a name, a type, and some attributes.

Each assembly source file contains at least one section. The number of sections in an assembly source file is only limited by the amount of memory available on the system at assembly time. If several sections with the same name are detected inside of a single source file, the code is concatenated into one large section.

Sections from different modules, but with the same name, will be combined into a single section at linking time.

Sections are defined through Section attributes on page 241 and Section types on page 242. The last part of the chapter deals with the merits of using relocatable sections. (See "Relocatable vs. absolute sections" on page 247.)

Section attributes

An attribute is associated with each section according to its content. A section may be:

- a data section,
- a constant data section, or
- a code section.

Code sections

A section containing at least one instruction is considered to be a code section. Code sections are always allocated in the target processor's ROM area.

Code sections should not contain any variable definitions (variables defined using the DS directive). You do not have any write access on variables defined in a code section. In addition, variables in code sections cannot be displayed in the debugger as data.

Constant sections

A section containing only constant data definition (variables defined using the DC or DCB directives) is considered to be a constant section. Constant sections should be allocated in the target processor's ROM area, otherwise they cannot be initialized at application loading time.

Data sections

A section containing only variables (variables defined using the `DS` directive) is considered to be a data section. Data sections are always allocated in the target processor’s RAM area.

NOTE A section containing variables (`DS`) and constants (`DC`) or code is not a data section. The default for such a section with mixed DC and code content is to put that content into ROM.

We strongly recommend that you use separate sections for the definition of variables and constant variables. This will prevent problems in the initialization of constant variables.

Section types

First of all, you should decide whether to use relocatable or absolute code in your application. The Assembler allows the mixing of absolute and relocatable sections in a single application and also in a single source file. The main difference between absolute and relocatable sections is the way symbol addresses are determined.

This section covers these two types of sections:

- Absolute sections on page 242
- Relocatable sections on page 244

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 6.1 on page 242 for an example of constructing absolute sections using the `ORG` assembler directive.

Listing 6.1 Example source code using ORG for absolute sections

```
      XDEF  entry
      ORG   $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 6.2 on page 243.

Listing 6.2 Example PRM file for Listing 6.1 on page 242

```
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 6.3 on page 244 for an example using the `SECTION` directive.

Listing 6.3 Example source code using SECTION for relocatable sections

```
          XDEF  entry
constSec: SECTION     ; Relocatable constant data section.
cst1:     DC.B  $A6
cst2:     DC.B  $BC

dataSec:  SECTION     ; Relocatable data section.
var:      DS.B  1

codeSec:  SECTION     ; Relocatable code section.
entry:
          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 6.4 on page 245.

Listing 6.4 PRM file for Listing 6.3 on page 244 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 6.5 on page 246:

Listing 6.5 PRM file for Listing 6.3 on page 244 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 6.5 on page 246,

- the `dataSec` section is allocated starting at `0x0100`.
- the `constSec` section is allocated starting at `0x8000`.
- the `codeSec` section is allocated starting at `0x8010`.
- 64 bytes of RAM are allocated in the stack starting at `0x0200`.

Relocatable vs. absolute sections

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

Modularity

An application is more modular when programming can be divided into smaller units called sections. The sections themselves can be distributed among different source files.

Multiple developers

When an application is split over different files, multiple developers can be involved in the development of the application. To avoid major problems when merging the different files, attention must be paid to the following items:

- An include file must be available for each assembly source file, containing `XREF` directives for each exported variable, constant and function. In addition, the interface to the function should be described there (parameter passing rules as well as the function return value).
- When accessing variables, constants, or functions from another module, the corresponding include file must be included.
- Variables or constants defined by another developer must always be referenced by their names.
- Before invoking a function implemented in another file, the developer should respect the function interface, i.e., the parameters are passed as expected and the return value is retrieved correctly.

Early development

The application can be developed before the application memory map is known. Often the application’s definitive memory map can only be determined once the size required for code and data can be evaluated. The size required for code or data can only be quantified once the major part of the application is implemented. When absolute sections are used, defining the definitive memory map is an iterative process of mapping and remapping the code. The assembly files must be edited, assembled, and linked several times. When relocatable sections are used, this can be achieved by editing the PRM file and linking the application.

Enhanced portability

As the memory map is not the same for each derivative (MCU), using relocatable sections allow easy porting of the code for another MCU. When porting relocatable code to another target you only need to link the application again with the appropriate memory map.

Tracking overlaps

When using absolute sections, the programmer must ensure that there is no overlap between the sections. When using relocatable sections, the programmer does not need to be concerned about any section overlapping another. The labels’ offsets are all evaluated relatively to the beginning of the section. Absolute addresses are determined and assigned by the linker.

Reusability

When using relocatable sections, code implemented to handle a specific I/O device (serial communication device), can be reused in another application without any modification.

7

Assembler Syntax

An assembler source program is a sequence of source statements. Each source statement is coded on one line of text and can be either a:

- Comment line on page 249 or a
- Source line on page 249.

Comment line

A comment can occupy an entire line to explain the purpose and usage of a block of statements or to describe an algorithm. A comment line contains a semicolon followed by a text (Listing 7.1 on page 249). Comments are included in the assembly listing, but are not significant to the Assembler.

An empty line is also considered to be a comment line.

Listing 7.1 Examples of comments

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

... (non comments)
```

Source line

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

- a Label field on page 250,
- an Operation field on page 250,
- 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 sets on page 250
- a Directive on page 265 name, or
- a Macro on page 265 name.

Instruction sets

Executable instructions for the M68HC08 processor are defined in the “CPU08 Reference Manual”.

HC08 instruction set

Table 7.2 on page 259 presents an overview of the instructions available for the HC08:

Table 7.1 HC08 instruction set

Instruction	Addressing modes	Description
ADC	#<expression> <expression> <expression>,X ,X <expression>,SP	Add with Carry
ADD	#<expression> <expression> <expression>,X ,X <expression>,SP	Add without carry
AIS	#<expression>	Add Immediate value (signed) to Stack Pointer
AIX	#<expression>	Add Immediate value (signed) to Index register H:X
AND	#<expression> <expression> <expression>,X ,X <expression>,SP	Logical AND
ASL	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Left
ASLA		Arithmetic Shift Left Accumulator
ASLX		Arithmetic Shift Left register X
ASR	<expression> <expression>,X ,X <expression>,SP	Arithmetic Shift Right
ASRA		Arithmetic Shift Right Accumulator
ASRX		Arithmetic Shift Right register X

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
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
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

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
BNE	<label>	Branch if Not Equal
BPL	<label>	Branch if Plus
BRA	<label>	Branch Always
BRCLR	BitNumber, <expression>, <label>	Branch if Bit is Clear
BRN	<label>	Branch Never
BRSET	BitNumber, <expression>, <label>	Branch if Bit Set
BSET	BitNumber,<expression>	Set Bit in memory
BSR	<label>	Branch to Subroutine
CBEQ	<expression>,<label> <expression>,X+,<label> X+,<label> <expression>,SP,<label>	Compare and Branch if Equal
CBEQA	#<expression>,<label>	
CBEQX	#<expression>,<label>	
CLC		Clear Carry bit
CLI		Clear Interrupt mask bit
CLR	<expression> <expression>,X ,X <expression>,SP	Clear memory
CLRA		Clear Accumulator A
CLRH		Clear index Register H
CLRX		Clear index Register X
CMP	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare accumulator with memory

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
COM	<expression> <expression>,X ,X <expression>,SP	One’s complement on memory location
COMA		One’s complement on accumulator A
COMX		One’s complement on register X
CPHX	#<expression> <expression> <expression>,SP	Compare index register H:X with memory Stack pointer and Extended addressing modes only available for HCS08 (-CS08 option)
CPX	#<expression> <expression> <expression>,X ,X <expression>,SP	Compare index register X with memory
DAA		Decimal Adjust Accumulator
DBNZ	<expression>,<label> <expression>,X,<label> X,<label> <expression>,SP,<label>	Decrement counter and Branch if Not Zero
DBNZA	<label>	
DBNZX	<label>	
DEC	<expression> <expression>,X ,X <expression>,SP	Decrement memory location
DECA		Decrement Accumulator
DECX		Decrement Index register
DIV		Divide

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
EOR	#<expression> <expression> <expression>,X ,X <expression>,SP	Exclusive OR Memory with accumulator
INC	<expression> ,X <expression>,X <expression>,SP	Increment memory location
INCA		Increment Accumulator
INCX		Increment register X
JMP	<expression> <expression>,X ,X	Jump to label
JSR	<expression> <expression>,X ,X	Jump to Subroutine
LDA	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Accumulator
LDHX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load Index register H:X from memory Indexed, Stack pointer and extended addressing modes are only available for HCS08 (-CS08 option).
LDX	#<expression> <expression> <expression>,X ,X <expression>,SP	Load index Register X from memory
LSL	<expression> <expression>,X ,X <expression>,SP	Logical Shift Left in memory

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
LSLA		Logical Shift Left Accumulator
LSLX		Logical Shift Left register X
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
PULA		Pull Accumulator from stack

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
PULH		Pull index register H from stack
PULX		Pull index register X from stack
ROL	<expression> <expression>,X ,X <expression>,SP	Rotate memory Left
ROLA		Rotate Accumulator Left
ROLX		Rotate register X Left
ROR	<expression> <expression>,X ,X <expression>,SP	Rotate memory Right
RORA		Rotate Accumulator Right
RORX		Rotate register X Right
RSP		Reset Stack Pointer
RTI		Return from Interrupt
RTS		Return from Subroutine
SBC	#<expression> <expression> <expression>,X ,X <expression>,SP	Subtract with Carry
SEC		Set Carry bit
SEI		Set Interrupt mask bit
STA	<expression> <expression>,X ,X <expression>,SP	Store Accumulator in Memory

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
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
TSTX		Test register X for negative or zero
TSX		Transfer SP to index register H:X
TXA		Transfer index register X to Accumulator

Table 7.1 HC08 instruction set (*continued*)

Instruction	Addressing modes	Description
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 7.4 on page 265):

Table 7.2 Special HC(S)08 instructions

Instruction	Addressing modes	Description
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 7.3 on page 260 presents an overview of the instructions available for the RS08.

Table 7.3 RS08 instructions

Instruction	Addressing Modes	Description
ADC	#<expression> <expression> ,X D[X] X	Add with Carry
ADCX		Alias for ADC X
ADD	#<expression> <expression> ,X D[X] X	Add without Carry
ADDX		Alias for ADD X
AND	#<expression> <expression> ,X D[X] X	Logical AND
ANDX		Alias for AND X
ASLA		Arithmetic Shift Left Accumulator (alias for LSLA)
BCC	<label>	Branch if Carry Bit Clear
BCLR	BitNumber, <expression> BitNumber,D[X] BitNumber,X	Clear one Bit in Memory
BCS	<label>	Branch if Carry Bit Set
BEQ	<label>	Branch if Equal
BGND		Background
BHS	<label>	Branch if Higher or Same
BLO	<label>	Branch if Lower
BNE	<label>	Branch if Not Equal
BRN	<label>	Branch Never (Alias for BRA *+$2)

Table 7.3 RS08 instructions (*continued*)

Instruction	Addressing Modes	Description
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 7.3 RS08 instructions (*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 7.3 RS08 instructions (*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 7.3 RS08 instructions (*continued*)

Instruction	Addressing Modes	Description
SBC	#<expression> <expression> ,X D[X] X	Subtract with Carry
SBCX		Substract 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)

Table 7.3 RS08 instructions (*continued*)

Instruction	Addressing Modes	Description
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 (HC08 / HCS08)

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 7.4 on page 265):

Table 7.4 HC(S)08 addressing mode notation

Addressing Mode	Notation	Example
Inherent on page 266	No operands	RSP
Immediate on page 267	#<expression>	ADC #$01

Table 7.4 HC(S)08 addressing mode notation (*continued*)

Addressing Mode	Notation	Example
Direct on page 267	<expression>	ADC byte
Extended on page 268	<expression>	ADC word
Indexed, no offset on page 268	,X	ADC ,X
IIndexed, 8-bit offset on page 269t	<expression>,X	ADC Offset,X
Indexed, 16-bit offset on page 270	<expression>,X	ADC Offset,X
Relative on page 270	<label>	BRA Label
Stack Pointer, 8-bit offset on page 271	<expression>,SP	ADC Offset,SP
Stack Pointer, 16-bit offset on page 271	<expression>,SP	ADC Offset,SP
Memory-to-memory immediate-to-direct on page 272	#<expression>,<expression>	MOV #$05,MyDataByte
Memory-to-memory direct-to-direct on page 272	<expression>,<expression>	MOV DatLoc1,DatLoc2
Memory-to-memory indexed-to-direct with post-increment on page 273t	X+,<expression>	MOV X+,<expression>
Memory-to-memory direct-to-indexed with post-increment on page 274	<expression>,X+	MOV <expression>,X+
Indexed with post-increment on page 274	X+	CBEQ X+, Data
Indexed, 8-bit offset, with post-increment on page 275	#<expression>,X+	CBEQ #offset,X+,Data

Inherent

Instructions using this addressing mode do not have any associated instruction fetch (Listing 7.2 on page 266). Some of them are acting on data in the CPU registers.

Listing 7.2 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 the Listing 7.3 on page 267.

Listing 7.3 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 7.4 on page 267 for an example of the direct addressing mode.

Listing 7.4 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 7.5 on page 268 for an example of the extended addressing mode.

Listing 7.5 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 7.6 on page 269 for an example of using the indexed (no offset) addressing mode.

Listing 7.6 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 7.7 on page 269 for an example of using the indexed (8-bit offset) addressing mode.

Listing 7.7 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 7.8 on page 270 for an example of using the indexed (16-bit offset) addressing mode.

Listing 7.8 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 7.9 on page 271 for an example of using the relative addressing mode.

Listing 7.9 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 7.10 on page 271 for an example of using the Stack Pointer *8-bit offset) addressing mode.

Listing 7.10 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 7.11 on page 271 for an example of using the Stack Pointer (16-bit offset) addressing mode.

Listing 7.11 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 7.12 on page 272 for an example for using the memory-to- memory immediate-to-direct addressing mode.

Listing 7.12 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 7.13 on page 272 for an example of using the memory-to- memory direct-to-direct addressing mode.

Listing 7.13 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 7.14 on page 273 for an example of using the memory-to-memory indexed to direct with post-increment addressing mode.

Listing 7.14 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 7.15 on page 274 for an example of using the memory-to-memory direct-to-indexed with post-increment addressing mode.

Listing 7.15 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 7.16 on page 275 for an example of an example of using the indexed with post-increment addressing mode.

Listing 7.16 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 7.17 on page 275 for an example of the indexed (8-bit offset) with post-increment addressing mode.

Listing 7.17 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 7.5 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

Table 7.5 Operand Field RS08 Addressing Modes (*continued*)

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 don’t have any instruction fetch associated. 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 on page 280
- External symbols on page 281
- Undefined symbols on page 282
- Reserved symbols on page 282

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 7.18 on page 281).

Listing 7.18 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 7.19 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 7.20 on page 281 for an example of using `XDEF` and `XREF`.

Listing 7.20 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 7.21 on page 282 shows an example of an undeclared label.

Listing 7.21 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 7.22 on page 282.

Listing 7.22 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. Also, PAGE is a reserved identifier.

Also, the keywords `MAP_ADDR_6` and `HIGH_6_13` are reserved identifiers. `HIGH_6_13` returns the higher byte for a given 14 bit address (used to load the PAGE register for the RS08). `MAP_ADDR_6` will return the lower 6 bits in a 14-bit address (used to determine the offset in the paging window for the RS08).

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 7.23 on page 283 for a variety of string constants.

Listing 7.23 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) on page 284
- Multiplication, division and modulo operators (binary) on page 285
- Sign operators (unary) on page 285
- Shift operators (binary) on page 286
- Bitwise operators (binary) on page 287
- Logical operators (unary) on page 288
- Relational operators (binary) on page 289
- HIGH operator on page 289
- on page 290HIGH_6_13 Operator on page 290
- LOW operator on page 290
- MAP_ADDR_6 Operator on page 291
- PAGE operator on page 292
- Force operator (unary) on page 292

Addition and subtraction operators (binary)

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

Syntax

```
Addition:    <operand> + <operand>
Subtraction: <operand> - <operand>
```

Description

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

Addition between two relocatable operands is not allowed.

Example

See Listing 7.24 on page 285 for an example of addition and subtraction operators.

Listing 7.24 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 7.25 on page 285 for an example of the multiplication, division, and modulo operators.

Listing 7.25 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 7.26 on page 286 for an example of the unary sign operators.

Listing 7.26 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 7.27 on page 287 for an example of the binary shift operators.

Listing 7.27 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 7.28 on page 287 for an example of the binary bitwise operators

Listing 7.28 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 7.29 on page 288 for an example of the unary bitwise operator.

Listing 7.29 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 7.30 on page 289 for an example of the unary logical operator.

Listing 7.30 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 7.31 on page 289 for an example of the binary relational operators

Listing 7.31 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 baseof 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 the operand to be an 8-bit operand, whereas the > or .W operators force the operand to be a 16-bit operand.

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

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

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

Example

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

For the RS08 the < operand forces the operand to short or tiny addressing mode (depending on the instruction where it is used). The same result can be obtained by adding .S or .T to the reffered symbol. The > operator may be used to force an address to 8 bits, even if it would fit in 4 or 5 bits (so short or tiny addressing modes could be used).

Operator precedence

Operator precedence follows the rules for ANSI - C operators (Table 7.6 on page 293)

Table 7.6 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

Table 7.6 Operator precedence priorities (*continued*)

Operator	Description	Associativity
<< >>	Shift Left Shift Right	Left to Right
< <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to Right
=, == !=, <>	Equal to Not Equal to	Left to Right
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
|	Bitwise OR	Left to Right

Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols
- External symbols
- The special symbol '*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the `DC` directive:

```
DC.W  1, 2, *-2
```

Once a valid expression has been fully evaluated by the Assembler, it is reduced as one of the following type of expressions:

- Absolute expression on page 295: The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant. Simple relocatable expression on page 296: The expression evaluates to an absolute offset from the start of a single relocatable section.
- Complex relocatable expression: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The Assembler does not support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in XREF directives, which are assumed to be relocatable symbols.

Absolute expression

An absolute expression is an expression involving constants or known absolute labels or expressions. An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

See Listing 7.32 on page 295 for an example of an absolute expression.

Listing 7.32 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 7.33 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 7.34 on page 295):

Listing 7.34 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 7.35 on page 296, the expression "tabEnd-tabBegin" evaluates to an absolute expression and is assigned the value of the difference between the offset of tabEnd and tabBegin in the section DataSec.

Listing 7.35 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 7.36 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 7.7 on page 297 describes the type of an expression according to the operator in an unary operation:

Table 7.7 Expression type resulting from operator and operand type

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

Binary operations result

Table 7.8 on page 297 describes the type of an expression according to the left and right operators in a binary operation:

Table 7.8 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`.

8

Assembler Directives

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

Directive overview

Section-Definition directives

The directives in Table 8.1 on page 299 are used to define new sections.

Table 8.1 Directives for defining sections

Directive	Description
ORG - Set Location Counter on page 353	Define an absolute section
SECTION - Declare Relocatable Section on page 361	Define a relocatable section
OFFSET - Create absolute symbols on page 351	Define an offset section

Constant-Definition directives

The directives in Table 8.2 on page 299 are used to define assembly constants.

Table 8.2 Directives for defining constants

Directive	Description
EQU - Equate symbol value on page 322	Assign a name to an expression (cannot be redefined)
SET - Set Symbol Value on page 363	Assign a name to an expression (can be redefined)

Data-Allocation directives

The directives in Table 8.3 on page 300 are used to allocate variables.

Table 8.3 Directives for allocating variables

Directive	Description
DC - Define Constant on page 311	Define a constant variable
DCB - Define Constant Block on page 313	Define a constant block
DS - Define Space on page 314	Define storage for a variable
RAD50 - Rad50-encoded string constants on page 358	RAD50 encoded string constants

Symbol-Linkage directives

Symbol-linkage directives (Table 8.4 on page 300) are used to export or import global symbols.

Table 8.4 Symbol linkage directives

Directive	Description
ABSENTRY - Application entry point on page 304	Specify the application entry point when an absolute file is generated
XDEF - External Symbol Definition on page 367	Make a symbol public (visible from outside)
XREF - External Symbol Reference on page 368	Import reference to an external symbol.
XREFB - External Reference for Symbols located on the Direct Page on page 369	Import reference to an external symbol located on the direct page.

Assembly-Control directives

Assembly-control directives (Table 8.5 on page 300) are general purpose directives used to control the assembly process.

Table 8.5 Assembly control directives

Directive	Description
ALIGN - Align Location Counter on page 306	Define Alignment Constraint
BASE - Set number base on page 307	Specify default base for constant definition
END - End assembly on page 318	End of assembly unit
ENDFOR - End of FOR block on page 319	End of FOR block
EVEN - Force word alignment on page 323	Define 2-byte alignment constraint
FAIL - Generate Error message on page 325	Generate user defined error or warning messages
FOR - Repeat assembly block on page 329	Repeat assembly blocks
INCLUDE - Include text from another file on page 335	Include text from another file.
LONGEVEN - Forcing Long-Word alignment on page 340	Define 4 Byte alignment constraint

Listing-File Control directives

Listing-file control directives (Table 8.6 on page 301) control the generation of the assembler listing file.

Table 8.6 Listing-file control directives

Directive	Description
CLIST - List conditional assembly on page 309	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable Listing on page 336	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set Line Length on page 338	Define line length in assembly listing file.

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

Directive	Description
MLIST - List macro expansions on page 345	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable Listing on page 348	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable Paging on page 350	Disable paging in the assembly listing file.
PAGE - Insert Page break on page 355	Insert page break.
PLEN - Set Page Length on page 357	Define page length in the assembler listing file.
SPC - Insert Blank Lines on page 364	Insert an empty line in the assembly listing file.
TABS - Set Tab Length on page 365	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide Listing Title on page 366	Define the user defined title for the assembler listing file.

Macro Control directives

Macro control directives (Table 8.7 on page 302) are used for the definition and expansion of macros.

Table 8.7 Macro control directives

Directive	Description
ENDM - End macro definition on page 321	End of user defined macro.
MACRO - Begin macro definition on page 341	Start of user defined macro.
MEXIT - Terminate Macro Expansion on page 342	Exit from macro expansion.

Conditional Assembly directives

Conditional assembly directives (Table 8.8 on page 302) are used for conditional assembling.

Table 8.8 Conditional assembly directives

Directive	Description
ELSE - Conditional assembly on page 316	alternate block
-Compat: Compatibility modes on page 319	End of conditional block
IF - Conditional assembly on page 331	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly on page 333	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined
IFNE	Test if an expression is not null.

Detailed descriptions of all assembler directives

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

ABSENTRY - Application entry point

Syntax

```
ABSENTRY <label>
```

Synonym

None

Description

This directive is used to specify the application Entry Point when the Assembler directly generates an absolute file. The `-FA2` assembly option - ELF/DWARF 2.0 Absolute File - must be enabled.

Using this directive, the entry point of the assembly application is written in the header of the generated absolute file. When this file is loaded in the debugger, the line where the entry point label is defined is highlighted in the source window.

This directive is ignored when the Assembler generates an object file.

NOTE This instruction only affects the loading on an application by a debugger. It tells the debugger which initial PC should be used. In order to start the application on a target - initialize the Reset vector.

If the example in Listing 8.1 on page 304 is assembled using the -FA2 assembler option, an ELF/DWARF 2.0 Absolute file is generated.

Listing 8.1 Using ABSENTRY to specify an application entry point

```
       ABSENTRY entry

       ORG   $fffe
Reset: DC.W  entry
       ORG   $70
entry: NOP
       NOP
main:  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 8.2 on page 306 aligns the `HEX` label to a location, which is a multiple of 16 (in this case, location `00010 (Hex)`)

Listing 8.2 Aligning the HEX Label to a Location

```
Assembler

Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1
    2    2   000000 6869 6768         DC.B  "high"
    3    3   000004 0000 0000         ALIGN 16
             000008 0000 0000
             00000C 0000 0000
    4    4
    5    5
    6    6   000010 7F          HEX:   DC.B 127 ; HEX is allocated
    7    7                                      ; on an address,
    8    8                                      ; which is a
    9    9                                      ; multiple of 16.
```

BASE - Set number base

Syntax

```
BASE <n>
```

Synonym

None

Description

The directive sets the default number base for constants to `<n>`. The operand `<n>` may be prefixed to indicate its number base; otherwise, the operand is considered to be in the current default base. Valid values of `<n>` are 2, 8, 10, 16. Unless a default base is specified using the `BASE` directive, the default number base is decimal.

Example

See Listing 8.3 on page 307 for examples of setting the number base.

Listing 8.3 Setting the number base

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

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

CLIST - List conditional assembly

Syntax

```
CLIST [ON|OFF]
```

Synonym

None

Description

The CLIST directive controls the listing of subsequent conditional assembly blocks. It precedes the first directive of the conditional assembly block to which it applies, and remains effective until the next CLIST directive is read.

When the ON keyword is specified in a CLIST directive, the listing file includes all directives and instructions in the conditional assembly block, even those which do not generate code (which are skipped).

When the OFF keyword is entered, only the directives and instructions that generate code are listed.

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

Example

Listing 8.4 on page 309 is an example where the CLIST OFF option is used.

Listing 8.4 Listing file with CLIST OFF

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

Listing 8.5 on page 310 is the corresponding listing file.

Listing 8.5 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 8.6 on page 310 is a listing file where `CLIST ON is used.`

Listing 8.6 CLIST ON is selected

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

Listing 8.7 on page 310 is the corresponding listing file.

Listing 8.7 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 8.8 on page 311).
- `DC.W`: Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary (Listing 8.9 on page 312).
- `DC.L`: Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary (Listing 8.10 on page 312).

Listing 8.8 Example for DC.B

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

Listing 8.9 Example for DC.W

```
000000 0041 4243   Label: DC.W "ABCDE"
000004 4445
000006 000A 000A          DC.W %1010, @12, 1, $A
00000A 0001 000A
00000E xxxx               DC.W Label
```

Listing 8.10 Example for DC.L

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445
000008 0000 000A          DC.L %1010, @12, 1, $A
00000C 0000 000A
000010 0000 0001
000014 0000 000A
000018 xxxx xxxx          DC.L Label
```

If the value in an operand expression exceeds the size of the operand, the value is truncated and a warning message is generated.

See also

Assembler directives:

- DCB - Define Constant Block on page 313
- DS - Define Space on page 314
- ORG - Set Location Counter on page 353
- SECTION - Declare Relocatable Section on page 361

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 8.11 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 on page 311
- DS - Define Space on page 314
- ORG - Set Location Counter on page 353
- SECTION - Declare Relocatable Section on page 361

DS - Define Space

Syntax

```
[<label>:]  DS[.<size>] <count>
```

where `<size>` = `B` (default), `W`, or `L`.

Synonym

```
RMB (= DS.B)
RMD (2 bytes)
RMQ (4 bytes)
```

Description

The `DS` directive is used to reserve memory for variables (Listing 8.12 on page 314). 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 8.12 Examples of DS directives

```
Counter:  DS.B  2 ; 2 continuous bytes in memory
          DS.B  2 ; 2 continuous bytes in memory
                  ; can only be accessed through the label Counter
          DS.W  5 ; 5 continuous words in memory
```

The label `Counter` references the lowest address of the defined storage area.

NOTE Storage allocated with a `DS` directive may end up in constant data section or even in a code section, if the same section contains constants or code as well. The Assembler allocates only a complete section at once.

Example

In Listing 8.13 on page 315 on page 315, a variable, a constant, and code were put in the same section. Because code has to be in ROM, then all three elements must be put into ROM. In order to allocate them separately, put them in different sections (Listing 8.14 on page 315).

Listing 8.13 Poor memory allocation

```
; How it should NOT be done ...
Counter:        DS 1     ; 1-byte used
InitialCounter: DC.B $f5 ; constant $f5
main:           NOP      ; NOP instruction
```

Listing 8.14 How it should be done...

```
DataSect:      SECTION    ; separate section for variables
Counter:       DS 1       ; 1-byte used

ConstSect:     SECTION  ; separate section for constants
InitialCounter: DC.B $f5  ; constant $f5

CodeSect:      SECTION    ; section for code
main:           NOP       ; NOP instruction
```

An `ORG` directive also starts a new section.

See also

Assembler directives:

- DC - Define Constant on page 311
- ORG - Set Location Counter on page 353
- SECTION - Declare Relocatable Section on page 361

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 8.15 on page 316 is an example of the use of conditional assembly directives:

Listing 8.15 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 8.16 Output listing of Listing 8.15 on page 316

```
 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 8.17 on page 318 causes any source code after the END statement to be ignored, as in Listing 8.18 on page 318.

Listing 8.17 Source File

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

Listing 8.18 Generated listing file

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 1234        Label:  DC.W  $1234
    2    2   000002 5678                DC.W  $5678
```

ENDFOR - End of FOR block

Syntax

```
ENDFOR
```

Synonym

None

Description

The `ENDFOR` directive indicates the end of a `FOR` block.

NOTE The `FOR` directive is only available when the `-Compat=b` assembler option is used. Otherwise, the `FOR` directive is not supported.

Example

See Listing 8.28 on page 329 in the `FOR`.section.

See also

Assembler directives:

- FOR - Repeat assembly block on page 329
- -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 8.30 on page 331 in the `IF` section.

See also

IF - Conditional assembly on page 331 assembler directive

ENDM - End macro definition

Syntax

```
ENDM
```

Synonym

None

Description

The `ENDM` directive terminates the macro definition (Listing 8.19 on page 321).

Example

The ENDM statement in Listing 8.19 on page 321 terminates the cpChar macro.

Listing 8.19 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 8.20 on page 322 for examples of using the `EQU` directive.

Listing 8.20 Using EQU to set variables

```
0000 0014  MaxElement: EQU  20
0000 0050  MaxSize:    EQU  MaxElement * 4

           Time:   DS.B  3
0000 0000  Hour:   EQU   Time   ; first byte addr.
0000 0002  Minute: EQU   Time+1 ; second byte addr
0000 0004  Second: EQU   Time+2 ; third byte addr
```

EVEN - Force word alignment

Syntax

```
EVEN
```

Synonym

None

Description

This directive forces the next instruction to the next even address relative to the start of the section. `EVEN` is an abbreviation for `ALIGN` 2. Some processors require word and long word operations to begin at even address boundaries. In such cases, the use of the `EVEN` directive ensures correct alignment. Omission of this directive can result in an error message.

Example

See Listing 8.21 on page 323 for instances where the EVEN directive causes padding bytes to be inserted.

Listing 8.21 Using the Force Word Alignment Directive

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000                   ds.b  4
    2    2                      ; location count has an even value
    3    3                      ; no padding byte inserted.
    4    4                            even
    5    5   000004                   ds.b  1
    6    6                      ; location count has an odd value
    7    7                      ; one padding byte inserted.
    8    8   000005                   even
    9    9   000006                   ds.b  3
   10   10                      ; location count has an odd value
   11   11                      ; one padding byte inserted.
   12   12   000009                   even
   13   13          0000 000A   aaa:  equ   10
```

See also

ALIGN - Align Location Counter on page 306 assembly directive

FAIL - Generate Error message

Syntax

```
FAIL  <arg>|<string>
```

Synonym

None

Description

There are three modes of the `FAIL` directive, depending upon the operand that is specified:

- If `<arg>` is a number in the range [`0–499`], the Assembler generates an error message, including the line number and argument of the directive. The Assembler does not generate an object file.
- If `<arg>` is a number in the range [`500–$FFFFFFFF`], the Assembler generates a warning message, including the line number and argument of the directive.
- If a string is supplied as an operand, the Assembler generates an error message, including the line number and the `<string>`. The Assembler does not generate an object file.
- The `FAIL` directive is primarily intended for use with conditional assembly to detect user-defined errors or warning conditions.

Examples

The assembly code in Listing 8.22 on page 325 generates the error messages in Listing 8.23 on page 326. The value of the operand associated with the ‘`FAIL  200`’ or ‘`FAIL  600`’ directives determines (1) the format of any warning or error message and (2) whether the source code segment will be assembled.

Listing 8.22 Example source code

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

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

Listing 8.23 Error messages resulting from assembling the source code in Listing 8.22 on page 325

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

          IFC "\2", ""
            FAIL   600
                  ^
WARNING A2332: FAIL found
Macro Call :             FAIL 600
```

Listing 8.24 on page 326 is another assembly code example which again incorporates the ‘`FAIL  200`’ and the ‘`FAIL  600`’ directives. Listing 8.25 on page 327 is the error message that was generated as a result of assembling the source code in Listing 8.24 on page 326.

Listing 8.24 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 8.25 Error messages resulting from assembling the source code in Listing 8.24 on page 326

```
>> in "C:\Freescale\demo\errfail.asm", line 6, col 19, pos 96

          IFC "\1", ""
            FAIL  200
                  ^
ERROR A2329: FAIL found
Macro Call :             FAIL 200
```

Listing 8.26 on page 327 has additional uses of the FAIL directive. In this example, the ‘FAIL string’ and ‘FAIL 600’ directives are used. Any error messages generated from the assembly code as a result of the FAIL directive are listed in Listing 8.27 on page 328.

Listing 8.26 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 8.27 Error messages resulting from assembling the source code in Listing 8.26 on page 327

```
>> in "C:\Freescale\demo\failmes.asm", line 7, col 17, pos 110

          IFC "\1", ""
            FAIL "A character must be specified as first parameter"
                   ^

ERROR A2338: A character must be specified as first parameter
Macro Call :   FAIL "A character must be specified as first parameter"
```

FOR - Repeat assembly block

Syntax

```
FOR <label>=<num> TO <num>
ENDFOR
```

Synonym

None

Description

The `FOR` directive is an inline macro because it can generate multiple lines of assembly code from only one line of input code.

`FOR` takes an absolute expression and assembles the portion of code following it, the number of times represented by the expression. The `FOR` expression may be either a constant or a label previously defined using `EQU` or `SET`.

NOTE The `FOR` directive is only available when the -Compat=b assembly option is used. Otherwise, the `FOR` directive is not supported.

Example

Listing 8.28 on page 329 is an example of using `FOR` to create a 5-repetition loop.

Listing 8.28 Using the FOR directive in a loop

```
FOR label=2 TO 6
  DC.B  label*7
ENDFOR
```

Listing 8.29 Resulting output listing

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      FOR label=2 TO 6
   2    2                         DC.B  label*7
   3    3                      ENDFOR
```

```
    4     2    000000 0E                   DC.B   label*7
    5     3                              ENDFOR
    6     2    000001 15                   DC.B   label*7
    7     3                              ENDFOR
    8     2    000002 1C                   DC.B   label*7
    9     3                              ENDFOR
   10     2    000003 23                   DC.B   label*7
   11     3                              ENDFOR
   12     2    000004 2A                   DC.B   label*7
   13     3                              ENDFOR
```

See also

on page 319ENDFOR - End of FOR block on page 319

-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 8.30 on page 331 is an example of the use of conditional assembly directives

Listing 8.30 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 8.31 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 8.9 on page 333 lists the available conditional types:

Table 8.9 Conditional assembly types

Ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>

Table 8.9 Conditional assembly types (*continued*)

Ifcc	Condition	Meaning
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Example

Listing 8.32 on page 334 is an example of the use of conditional assembler directives:

Listing 8.32 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 8.33 on page 334 shows the listing provided by the Assembler for these lines of code

Listing 8.33 output listing for Listing 8.32 on page 334

```
    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 on page 348 or an END - End assembly on page 318 assembler directive is reached

This directive is not written to the listing and debug files.

Example

The assembly source code using the `LIST` and `NOLIST` directives in Listing 8.34 on page 336 generates the output listing in Listing 8.35 on page 337.

Listing 8.34 Using the LIST and NOLIST assembler directives

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP         NOP
```

Listing 8.35 Output listing generated from running Listing 8.34 on page 336

```
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 8.37 on page 338 generates the listing file in Listing 8.37 on page 338. Notice that the '`LLEN 24`' directive causes the output at the location-counter line 7 to be truncated.

Listing 8.36 Example assembly source code using LLEN

```
         DC.B   $55
         LLEN   32
         DC.W   $1234, $4567

         LLEN   24
         DC.W   $1234, $4567
         EVEN
```

Listing 8.37 Formatted assembly output listing as a result of using LLEN

```
Abs. Rel.    Loc     Obj. code    Source line
---- ----    ------ ---------    -----------
   1    1    000000 55                    DC.B   $55
```

```
    2    2
    4    4   000001 1234 4567           DC.W  $1234, $4567
    5    5
    7    7   000005 1234 4567           DC.W  $1234, $
    8    8   000009 00                  EVEN
```

LONGEVEN - Forcing Long-Word alignment

Syntax

```
LONGEVEN
```

Synonym

None

Description

This directive forces the next instruction to the next long-word address relative to the start of the section. `LONGEVEN` is an abbreviation for `ALIGN` 4.

Example

See Listing 8.38 on page 340 for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

Listing 8.38 Forcing Long Word Alignment

```
    2    2   000000 01                         dcb.b 1,1
              ; location counter is not a multiple of 4; three filling
              ; bytes are required.
    3    3   000001 0000 00                    longeven
    4    4   000004 0002 0002                  dcb.w 2,2
              ; location counter is already a multiple of 4; no filling
              ; bytes are required.
    5    5                                     longeven
    6    6   000008 0202                       dcb.b 2,2
    7    7    ; following is for text section
    8    8                        s27          SECTION 27
    9    9   000000 9D                         nop
              ; location counter is not a multiple of 4; three filling
              ; bytes are required.
   10   10   000001 0000 00                    longeven
   11   11   000004 9D                         nop
```

MACRO - Begin macro definition

Syntax

`<label>: MACRO`

Synonym

None

Description

The `<label>` of the `MACRO` directive is the name by which the macro is called. This name must not be a processor machine instruction or assembler directive name. For more information on macros, see the *Macros* chapter.

Example

See Listing 8.39 on page 341 for a macro definition.

Listing 8.39 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 on page 321 directive.

Example

See Listing 8.40 on page 342 allows the replication of simple instructions or directives using MACRO with MEXIT.

Listing 8.40 Example assembly code using MEXIT

```
         XDEF   entry

storage: EQU    $00FF

save:    MACRO            ; Start macro definition
         LDX    #storage
         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 8.41 on page 343 shows the macro expansion of the previous macro.

Listing 8.41 Macro expansion of Listing 8.40 on page 342

```
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 8.42 on page 345, with `MLIST ON`, generates the assembler output listing in Listing 8.43 on page 346

Listing 8.42 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 8.43 Assembler output listing of the example in Listing 8.42 on page 345 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 8.44 on page 346.

Listing 8.44 Assembler output listing of the example in Listing 8.42 on page 345 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 on page 336 assembler directive is reached.

Example

See Listing 8.45 on page 348 for an example of using `LIST` and `NOLIST`.

Listing 8.45 Examples of LIST and NOLIST

```
aaa:    NOP

        LIST
bbb:    NOP
        NOP

        NOLIST
ccc:    NOP
        NOP

        LIST
ddd:    NOP
        NOP
```

The listing above generates the listing file in Listing 8.46 on page 349.

Listing 8.46 Assembler output listing from the assembler source code in Listing 8.45 on page 348

```
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 on page 336 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 8.47 on page 351 shows how the `OFFSET` directive can be used to access an element of a structure.

Listing 8.47 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 8.48 on page 352).

Listing 8.48 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 8.49 on page 353 for an example where ORG sets the location counter.

Listing 8.49 Using ORG to set the location counter

```
      org   $2000
b1:   nop
b2:   rts
```

Viewing Listing 8.50 on page 353, you can see that the b1 label is located at address $2000 and label b2 is at address $2001.

Listing 8.50 Assembler output listing from the source code in Listing 8.49 on page 353

```
 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 on page 311
- DCB - Define Constant Block on page 313
- DS - Define Space on page 314
- SECTION - Declare Relocatable Section on page 361

PAGE - Insert Page break

Syntax

```
PAGE
```

Synonym

None

Description

Insert a page break in the assembly listing.

Example

The portion of code in Listing 8.51 on page 355 demonstrates the use of a page break in the assembler output listing.

Listing 8.51 Example assembly source code

```
code:   SECTION
        DC.B  $00,$12
        DC.B  $00,$34
        PAGE
        DC.B  $00,$56
        DC.B  $00,$78
```

The effect of the PAGE directive can be seen in Listing 8.52 on page 355.

Listing 8.52 Assembler output listing from the source code in Listing 8.51 on page 355

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      code:   SECTION
    2    2   000000 0012                DC.B  $00,$12
    3    3   000002 0034                DC.B  $00,$34


 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    5    5   000004 0056                DC.B    $00,$56
    6      6   000006 0078                  DC.B     $00,$78
```

PLEN - Set Page Length

Syntax

```
PLEN<n>
```

Synonym

None

Description

Sets the listings page length to `<n>` lines. `<n>` may range from `10` to `10000`. If the number of lines already listed on the current page is greater than or equal to `<n>`, listing will continue on the next page with the new page length setting.

The default page length is 65 lines.

RAD50 - Rad50-encoded string constants

Syntax

```
RAD50 <str>[, cnt]
```

Synonym

None

Description

This directive places strings encoded with the RAD50 encoding into constants. The RAD50 encoding places 3 string characters out of a reduced character set into 2 bytes. It therefore saves memory when comparing it with a plain ASCII representation. It also has some drawbacks, however. Only 40 different character values are supported, and the strings have to be decoded before they can be used. This decoding does include some computations including divisions (not just shifts) and is therefore rather expensive.

The encoding takes three bytes and looks them up in a string table (Listing 8.53 on page 358).

Listing 8.53 RAD50 encoding

```
unsigned short LookUpPos(char x) {
  static const char translate[]=
    " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";

  const char* pos= strchr(translate, x);
  if (pos == NULL) { EncodingError(); return 0; }
  return pos-translate;
}
unsigned short Encode(char a, char b, char c) {
  return LookUpPos(a)*40*40 + LookUpPos(b)*40
                            + LookUpPos(c);
}
```

If the remaining string is shorter than 3 bytes, it is filled with spaces (which correspond to the RAD50 character 0).

The optional argument `cnt` can be used to explicitly state how many 16-bit values should be written. If the string is shorter than `3*cnt`, then it is filled with spaces.

See the example C code below (Listing 8.56 on page 359) about how to decode it.

Example

The string data in Listing 8.54 on page 359 assembles to the following data (Listing 8.55 on page 359). The 11 characters in the string are represented by 8 bytes.

Listing 8.54 RAD50 Example

```
  XDEF rad50, rad50Len
DataSection    SECTION
rad50:         RAD50 "Hello World"
rad50Len:      EQU (*-rad50)/2
```

Listing 8.55 Assembler output where 11 characters are contained in eight bytes

```
$32D4 $4D58 $922A $4BA0
```

This C code shown in Listing 8.56 on page 359 takes the data and prints “Hello World”.

Listing 8.56 Example—Program that Prints Hello World

```
#include "stdio.h"
extern unsigned short rad50[];
extern int rad50Len; /* address is value. Exported asm label */
#define rad50len ((int) &rad50Len)

void printRadChar(char ch) {
  static const char translate[]=
    " ABCDEFGHIJKLMNOPQRSTUVWXYZ$.?0123456789";
  char asciiChar= translate[ch];
  (void)putchar(asciiChar);
}
void PrintHallo(void) {
  unsigned char values= rad50len;
  unsigned char i;
  for (i=0; i < values; i++) {
    unsigned short val= rad50[i];
    printRadChar(val / (40 * 40));
    printRadChar((val / 40) % 40);
    printRadChar(val % 40);
  }
}
```

SECTION - Declare Relocatable Section

Syntax

```
<name>: SECTION [SHORT] [RS08_SHORT] [RS08_TINY] [<number>]
```

Synonym

None

Description

This directive declares a relocatable section and initializes the location counter for the following code. The first SECTION directive for a section sets the location counter to zero. Subsequent SECTION directives for that section restore the location counter to the value that follows the address of the last code in the section.

<name> is the name assigned to the section. Two SECTION directives with the same name specified refer to the same section.

<number> is optional and is only specified for compatibility with the MASM Assembler.

A section is a code section when it contains at least one assembly instruction. It is considered to be a constant section if it contains only DC or DCB directives. A section is considered to be a data section when it contains at least a DS directive or if it is empty.

Example

The example in Listing 8.57 on page 361 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 8.57 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.

Example

The following example demonstrates the definition and usage of a `SHORT` section.

In the example shown in Listing 8.58 on page 362, the symbol data is accessed using the direct addressing mode.

Listing 8.58 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 on page 353
- DC - Define Constant on page 311
- DCB - Define Constant Block on page 313
- DS - Define Space on page 314

SET - Set Symbol Value

Syntax

```
<label>: SET <expression>
```

Synonym

None

Description

Similar to the EQU - Equate symbol value on page 322 directive, the `SET` directive assigns the value of the `<expression>` in the operand field to the symbol in the `<label>` field. The `<expression>` must resolve as an absolute expression and cannot include a symbol that is undefined or not yet defined. The `<label>` is an assembly time constant. `SET` does not generate any machine code.

The value is temporary; a subsequent `SET` directive can redefine it.

Example

See Listing 8.59 on page 363 for examples of the SET directive.

Listing 8.59 Using the SET assembler directive

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1          0000 0002   count:  SET   2
   2    2   000000 02          one:    DC.B  count
   3    3
   4    4          0000 0001   count:  SET   count-1
   5    5   000001 01                  DC.B  count
   6    6
   7    7          0000 0001           IFNE  count
   8    8          0000 0000   count:  SET   count-1
   9    9                              ENDIF
  10   10   000002 00                  DC.B  count
```

The value associated with the label `count` is decremented after each `DC.B` instruction.

SPC - Insert Blank Lines

Syntax

```
SPC<count>
```

Synonym

None

Description

Inserts `<count>` blank lines in the assembly listing. `<count>` may range from 0 to 65. This has the same effect as writing that number of blank lines in the assembly source. A blank line is a line containing only a carriage return.

TABS - Set Tab Length

Syntax

```
TABS <n>
```

Synonym

None

Description

Sets the tab length to `<n>` spaces. The default tab length is eight. `<n>` may range from 0 to 128.

TITLE - Provide Listing Title

Syntax

```
TITLE "title"
```

Synonym

```
TTL
```

Description

Print the `<title>` on the head of every page of the listing file. This directive must be the first source code line. A title consists of a string of characters enclosed in quotes `(").`

The title specified will be written on the top of each page in the assembly listing file.

XDEF - External Symbol Definition

Syntax

```
XDEF [.<size>] <label>[,<label>]...
```

where `<size> = B(direct), W` (default)`, or L 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 8.60 on page 367 for the case where the `XDEF` assembler directive can specify symbols that can be used by other modules.

Listing 8.60 Using XDEF to create a variable to be used in another file

```
        XDEF Count, main
        ;; variable Count can be referenced in other modules,
        ;; same for label main. Note that Linker & Assembler
        ;; are case-sensitive, i.e., Count != count.

Count:  DS.W  2

code:   SECTION
main:   DC.B 1
```

XREF - External Symbol Reference

Syntax

```
XREF [.<size>] <symbol>[,<symbol>]...
```

where `<size> = B`(direct), `W` (default), or `L`.

Synonym

```
EXTERNAL
```

Description

This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 32-bit values is passed to the linker.

The number of symbols enumerated in an `XREF` directive is only limited by the memory available at assembly time.

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.)
```

9

Macros

A macro is a template for a code sequence. Once a macro is defined, subsequent reference to the macro name are replaced by its code sequence.

Macro overview

A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently called.

The Assembler expands the macro definition each time the macro is called. The macro call causes source statements to be generated, which may include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Calling previously defined macros is also allowed. Source statements generated by a macro call are inserted in the source file at the position where the macro is invoked.

To call a macro, write the macro name in the operation field of a source statement. Place the arguments in the operand field. The macro may contain conditional assembly directives that cause the Assembler to produce in-line-coding variations of the macro definition.

Macros call produces in-line code to perform a predefined function. Each time the macro is called, code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program.

Defining a macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.
- The body of the macro, a sequential list of assembler statements, some possibly including argument placeholders.
- The `ENDM` directive, terminating the macro definition.
- eventually an instruction `MEXIT`, which stops macro expansion.

See the Assembler Directives chapter for information about the `MACRO`, `ENDM`, `MEXIT`, and `MLIST` directives.

The body of a macro is a sequence of assembler source statements. Macro parameters are defined by the appearance of parameter designators within these source statements. Valid macro definition statements includes the set of processor assembly language instructions, assembler directives, and calls to previously defined macros. However, macro definitions may not be nested.

Calling macros

The form of a macro call is:

```
[<label>:] <name>[.<sizearg>] [<argument> [,<argument>]...]
```

Although a macro may be referenced by another macro prior to its definition in the source module, a macro must be defined before its first call. The name of the called macro must appear in the operation field of the source statement. Arguments are supplied in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the call, according to the macro definition and the arguments specified in the macro call. The source statements of the expanded macro are then assembled subject to the same conditions and restrictions affecting any source statement. Nested macros calls are also expanded at this time.

Macro parameters

As many as 36 different substitutable parameters can be used in the source statements that constitute the body of a macro. These parameters are replaced by the corresponding arguments in a subsequent call to that macro.

A parameter designator consists of a backlashes character (\), followed by a digit (0 - 9) or an uppercase letter (A - Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Consider the macro definition in Listing 9.1 on page 372:

Listing 9.1 Example macro definition

```
MyMacro: MACRO
           DC.\0   \1, \2
         ENDM
```

When this macro is used in a program, e.g.:

```
MyMacro.B $10, $56
```

the Assembler expands it to:

```
    DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator `\1` through `\9` and `\A` through `\Z`, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument itself is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

Macro argument grouping

To pass text including commas as a single macro argument, the Assembler supports a special syntax. This grouping starts with the [? prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, brackets, question marks and backward slashes can also be escaped with a backward slash as 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 9.2 Example macro definition

```
MyMacro:  MACRO
            DC    \1
          ENDM
MyMacro1: MACRO
            \1
          ENDM
```

Listing 9.3 on page 374 has some macro calls with rather complicated arguments:

Listing 9.3 Macro calls for Listing 9.2 on page 373

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro  [?$10, $56?]?]
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines (Listing 9.4 on page 374):

Listing 9.4 Macro expansion of Listing 9.3 on page 374

```
DC    $10, $56
DC    "[?"
DC    $10, $56
DC    $10, $56
```

The Macro Assembler does also supports for compatibility with previous version’s macro grouping with an angle bracket syntax (Listing 9.5 on page 374):

Listing 9.5 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 9.6 on page 374) does not produce the expected result:

Listing 9.6 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 9.7 on page 375):

Listing 9.7 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 9.8 on page 375):

Listing 9.8 Calling the clear macro

```
         clear     temporary
         clear     data
```

The two macro calls of `clear` are expanded in the following manner (Listing 9.9 on page 375):

Listing 9.9 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 call are also supported.

A macro call is limited to the length of one line, i.e., 1024 characters.

10

Assembler Listing File

The assembly listing file is the output file of the Assembler that contains information about the generated code. The listing file is generated when the –L assembler option is activated. When an error is detected during assembling from the file, no listing file is generated.

The amount of information available depends upon the following assembler options:

- -L: Generate a listing file
- -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 10.1 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 on page 378:

Abs.

This column contains the absolute line number for each instruction. The absolute line number is the line number in the debug listing file, which contains all included files and where any macro calls have been expanded.

Listing 10.2 Example output listing - Abs. column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;-------------------------------
   2    2                      ; File: test.o
   3    3                      ;-------------------------------
   4    4
   5    5                               XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B   1
   8    8   000001             char2:   DS.B   1
   9    9                               INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                                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 10.3 on page 379.

An `'i'` suffix is appended to the relative line number when the line comes from an included file. An `'m'` suffix is appended to the relative line number when the line is generated by a macro call.

Listing 10.3 Example listing file - Rel. column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;-------------------------------
   2    2                      ; File: test.o
   3    3                      ;-------------------------------
   4    4
   5    5                               XDEF Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                               INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                                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 10.4 on page 380.

Listing 10.4 Example Listing File - Loc column

```
Abs. Rel.  Loc     Obj. code   Source line
---- ----  ------- ---------   -----------
   1    1                      ;------------------------------
   2    2                      ; File: test.o
   3    3                      ;------------------------------
   4    4
   5    5                               XDEF  Start
   6    6                      MyData:  SECTION
   7    7   000000             char1:   DS.B  1
   8    8   000001             char2:   DS.B  1
   9    9                               INCLUDE "macro.inc"
  10    1i                     cpChar:  MACRO
  11    2i                                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 10.5 on page 381.

Listing 10.5 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 10.6 on page 382.

Listing 10.6 Example listing file - Source line column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      ;------------------------------
   2    2                      ; File: test.o
   3    3                      ;------------------------------
   4    4
   5    5                               XDEF  Start
   6    6                      MyData: SECTION
   7    7   000000             char1:  DS.B  1
   8    8   000001             char2:  DS.B  1
   9    9                               INCLUDE "macro.inc"
  10   1i                      cpChar: MACRO
  11   2i                                 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
```

11

Mixed C and Assembler Applications

When you intend to mix Assembly source file and ANSI-C source files in a single application, the following issues are important:

- “Memory models” on page 383
- “Parameter passing scheme” on page 384
- “Return Value” on page 384
- “Accessing assembly variables in an ANSI-C source file” on page 384
- “Accessing ANSI-C variables in an assembly source file” on page 385
- “Invoking an assembly function in an ANSI-C source file” on page 386
- “Support for structured types” on page 389

To build mixed C and Assembler applications, you have to know how the C Compiler uses registers and calls procedures. The following sections will describe this for compatibility with the compiler. If you are working with another vendor’s ANSI-C compiler, refer to your Compiler Manual to get the information about parameter passing rules.

Memory models

The memory models are only important if you mix C and assembly code. In this case all sources must be compiled or assembled with the same memory model.

The Assembler supports all memory models of the compiler. Depending on your hardware, use the smallest memory model suitable for your programming needs.

Table 11.1 on page 384 summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

Table 11.1 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

Please check the compiler manual, back-end chapter about the details of parameter passing.

Return Value

Please check the compiler manual’s backend chapter about 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 11.1 on page 385).

Listing 11.1 Example of data and constant definition

```
          XDEF  ASMData, ASMConst
DataSec:  SECTION
ASMData:  DS.W  1      ; Definition of a variable
ConstSec: SECTION
ASMConst: DC.W  $44A6  ; Definition of a constant
```

We recommend that you generate a header file for each assembler source file. This header file should contain the interface to the assembly module.

An external declaration for the variable or constant must be inserted in the header file (Listing 11.2 on page 385).

Listing 11.2 Example of data and constant declarations

```
/* External declaration of a variable */
extern int       ASMData;
/* External declaration of a constant */
extern const int ASMConst;
```

The variables or constants can then be accessed in the usual way, using their names (Listing 11.3 on page 385).

Listing 11.3 Example of data and constant reference

```
  ASMData = ASMConst + 3;
```

Accessing ANSI-C variables in an assembly source file

A variable or constant defined in an ANSI-C source file is accessible in an assembly source file.

The variable or constant is defined in the ANSI-C source file using the standard ANSI-C syntax (Listing 11.4 on page 386).

Listing 11.4 Example definition of data and constants

```
unsigned int CData;        /* Definition of a variable */
unsigned const int CConst; /* Definition of a constant */
```

An external declaration for the variable or constant must be inserted into the assembly source file (Listing 11.5 on page 386).

This can also be done in a separate file, included in the assembly source file.

Listing 11.5 Example declaration of data and constants

```
XREF CData;  External declaration of a variable
XREF CConst; External declaration of a constant
```

The variables or constants can then be accessed in the usual way, using their names (Listing 11.6 on page 386).

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

Listing 11.6 Example of data and constant reference

```
           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 11.7 on page 387) can be invoked in a C source file (Listing 11.9 on page 388). During the implementation of the function in the assembly source file, you should pay attention to the parameter passing scheme of the ANSI-C compiler you are using in order to retrieve the parameter from the right place.

Listing 11.7 Example of an assembly file: mixasm.asm

```
         XREF CData
         XDEF AddVar
         XDEF ASMData

DataSec: SECTION
ASMData: DS.B 1
CodeSec: SECTION
AddVar:
         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 11.7 on page 387). This header file (`mixasm.h` in Listing 11.8 on page 387) should contain the interface to the assembly module.

Listing 11.8 Header file for the assembly mixasm.asm file: mixasm.h

```
/* mixasm.h */
#ifndef _MIXASM_H_
#define _MIXASM_H_

void AddVar(unsigned char value);
/* function that adds the parameter value to global CData */
/* and then stores the result in ASMData */

/* variable which receives the result of AddVar */
extern char ASMData;

#endif /* _MIXASM_H_ */
```

The function can then be invoked in the usual way, using its name.

Example of a C file

A C source code file (`mixc.c`) has the main() function which calls the `AddVar()` function. See Listing 11.9 on page 388. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format).

Listing 11.9 Example C source code file: mixc.c

```
static int Error          = 0;
const unsigned char CData = 12;
#include "mixasm.h"

void main(void) {
  AddVar(10);
  if (ASMData != CData + 10){
    Error = 1;
  } else {
    Error = 0;
  }
  for(;;); // wait forever
}
```

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 11.10 on page 388.

Listing 11.10 Example of linker parameter file: mixasm.prm

```
LINK mixasm.abs
NAMES
  mixc.o mixasm.o
END
SECTIONS
  MY_ROM   = READ_ONLY  0x4000 TO 0x4FFF;
  MY_RAM   = READ_WRITE 0x2400 TO 0x2FFF;
  MY_STACK = READ_WRITE 0x2000 TO 0x23FF;
END
PLACEMENT
  DEFAULT_RAM    INTO MY_RAM;
  DEFAULT_ROM    INTO MY_ROM;
  SSTACK         INTO MY_STACK;
END
INIT main
```

NOTE We recommend that you use the same memory model and object file format for all the generated object files.

Support for structured types

When the -Struct: Support for structured types assembler option is activated, the Macro Assembler also supports the definition and usage of structured types. This allows an easier way to access ANSI-C structured variable in the Macro Assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type. See “Structured type definition” on page 389.
- Define a structured variable. See “Variable definition” on page 391.
- Declare a structured variable. See “Variable declaration” on page 391.
- Access the address of a field inside of a structured variable. See “Accessing a field address” on page 392
- Access the offset of a field inside of a structured variable. See “Accessing a field offset” on page 393.

NOTE Some limitations apply in the usage of the structured types in the Macro Assembler. See Structured type: Limitations on page 393.

Structured type definition

The Macro Assembler is extended with the following new keywords in order to support ANSI-C type definitions.

- `STRUCT`
- `UNION`

The structured type definition for `STRUCT` can be encoded as in Listing 11.11 on page 389:

Listing 11.11 Definition for STRUCT

```
typeName: STRUCT
   lab1: DS.W  1   lab2: DS.W  1         ...
   ENDSTRUCT
```

where:

`'typeName'` is the name associated with the defined type. The type name is considered to be a user-defined keyword. The Macro Assembler will be case-insensitive on `typeName`.

`'STRUCT'` specifies that the type is a structured type.

`'lab1'`and `'lab2'` are the fields defined inside of the `'typeName' type`. The fields will be considered as user-defined labels, and the Macro Assembler will be case-sensitive on label names.

As with all other directives in the Assembler, the `STRUCT` and `UNION` directives are case-insensitive.

The `STRUCT` and `UNION` directives cannot start on column 1 and must be preceded by a label.

Types allowed for structured type fields

The field inside of a structured type may be:

- another structured type or
- a base type, which can be mapped on 1, 2, or 4 bytes.

Table 11.2 on page 390 shows how the ANSI-C standard types are converted in the assembler notation:

Table 11.2 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 11.12 on page 391):

```
var: typeName
```

where:

- `'var'` is the name of the variable.
- `'typeName'` is the type associated with the variable.

Listing 11.12 Assembly code analog of a C struct of type: myType

```
myType:   STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
          ENDSTRUCT

DataSection: SECTION
structVar:   TYPE myType ; var ‘structVar’ is of type ‘myType’
```

Variable declaration

The Macro Assembler can provide a way to associated a type with a symbol which is defined externally. This is done by extending the `XREF` syntax:

```
  XREF var: typeName, var2
```

where:

- '`var`' is the name of an externally defined symbol.
- '`typeName`' is the type associated with the variable 'var'.

`'var2'` is the name of another externally defined symbol. This symbol is not associated with any type. See Listing 11.13 on page 391 for an example.

Listing 11.13 Example of extending XREF

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
```

```
        ENDSTRUCT

        XREF extData: myType ; var ‘extData’ is type ‘myType’
```

Accessing a structured variable

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

Accessing a field address

To access a structured-type field address (Listing 11.14 on page 392), the Assembler uses the colon character ':'.

`var:field`

where

- '`var`' is the name of a variable, which was associated with a structured type.
- '`field`' is the name of a field in the structured type associated with the variable.

Listing 11.14 Example of accessing a field address

```
myType:   STRUCT
field1:     DS.W  1
field2:     DS.W  1
field3:     DS.B  1
field4:     DS.B  3
field5:     DS.W  1
          ENDSTRUCT

          XREF  myData:myType
          XDEF  entry

CodeSec:  SECTION
entry:
          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 11.15 on page 393 for an example of using this notation for accessing an offset.

Listing 11.15 Accessing a field offset with the -><field> notation

```
myType:  STRUCT
field1:    DS.W  1
field2:    DS.W  1
field3:    DS.B  1
field4:    DS.B  3
field5:    DS.W  1
         ENDSTRUCT
         XREF.B myData
         XDEF   entry

CodeSec: SECTION
entry:
           LDX #myData
           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.

12

Make Applications

This chapters has the following sections:

- “Assembly applications” on page 395
- “Memory maps and segmentation” on page 396

Assembly applications

This section covers:

- Directly generating an absolute file on page 395
- Mixed C and assembly applications on page 395

Directly generating an absolute file

When an absolute file is directly generated by the Assembler:

- the application entry point must be specified in the assembly source file using the directive `ABSENTRY`.
- The whole application must be encoded in a single assembly unit.
- The application should only contain absolute sections.

Generating object files

The entry point of the application must be mentioned in the Linker parameter file using the "`INIT` *funcname*" command. The application is build of the different object files with the Linker. The Linker is document in a separate document.

Your assembly source files must be separately assembled. Then the list of all the object files building the application must be enumerated in the application PRM file.

Mixed C and assembly applications

Normally the application starts with the main procedure of a C file. All necessary object files - assembly or C - are linked with the Linker in the same fashion like pure C applications. The Linker is documented in a separate document.

Memory maps and segmentation

Relocatable Code Sections are placed in the `DEFAULT_ROM` or `.text` Segment.

Relocatable Data Sections are placed in the `DEFAULT_RAM` or `.data` Segment.

NOTE The `.text` and `.data` names are only supported when the ELF object file format is used.

There are no checks at all that variables are in `RAM`. If you mix code and data in a section you cannot place the section into `ROM`. That is why we suggest that you separate code and data into different sections.

If you want to place a section in a specific address range, you have to put the section name in the placement portion of the linker parameter file (Listing 12.1 on page 396).

Listing 12.1 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
```

13

How to ...

This chapter covers the following topics:

How to work with absolute sections

An absolute section is a section whose start address is known at assembly time.

(See modules fiboorg.asm and fiboorg.prm in the demo directory)

Defining absolute sections in an assembly source file

An absolute section is defined using the ORG directive. In that case, the Macro Assembler generates a pseudo section, whose name is "ORG_<index>", where index is an integer which is incremented each time an absolute section is encountered (Listing 13.1 on page 397).

Listing 13.1 Defining an absolute section containing data

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

In the previous portion of code, the label cst1 is located at address $A00, and label cst2 is located at address $A01.

Listing 13.2 Assembler output listing for Listing 13.1 on page 397

```
1     1                           ORG    $800
2     2  a000800        var:      DS.B   1
3     3                           ORG    $A00
4     4  a000A00 A6     cst1:     DC.B   $A6
5     5  a000A01 BC     cst2:     DC.B   $BC
```

Program assembly source code should be located in a separate absolute section (Listing 13.3 on page 398).

Listing 13.3 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 13.4 on page 398.

Listing 13.4 Assembler output listing for Listing 13.3 on page 398

```
    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 13.5 on page 399.

Listing 13.5 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.

How to work with relocatable sections

A relocatable section is a section which start address is determined at linking time.

Defining relocatable sections in a source file

A relocatable section is defined using the `SECTION` directive. See Listing 13.6 on page 400 for an example of defining relocatable sections.

Listing 13.6 Defining relocatable sections containing data

```
constSec: SECTION   ; Relocatable constant data section.
cst1:     DC.B  $A6
cst2:     DC.B  $BC

dataSec:  SECTION   ; Relocatable data section.
var:      DS.B  1
```

In the previous portion of code, the label `cst1` will be located at an offset 0 from the section `constSec` start address, and label `cst2` will be located at an offset 1 from the section `constSec` start address. See Listing 13.7 on page 400.

Listing 13.7 Assembler output listing for Listing 13.6 on page 400

```
  2    2                       constSec: SECTION ; Relocatable
  3    3   000000 A6           cst1:     DC.B    $A6
  4    4   000001 BC           cst2:     DC.B    $BC
  5    5
  6    6                       dataSec:  SECTION ; Relocatable
```

```
   7     7   000000                 var:       DS.B   1
```

Program assembly source code should be located in a separate relocatable section (Listing 13.8 on page 401Listing 13.8 on page 401).

Listing 13.8 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 13.9 on page 402.

Listing 13.9 Minimal linker parameter file

```
/* Name of the executable file generated.      */
LINK test.abs
/* Name of the object file in the application. */
NAMES
  test.o
END
SECTIONS
/* READ_ONLY memory area.  */
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.          */
  DEFAULT_RAM              INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM, constSec    INTO MY_ROM;
END
INIT entry                  /* Application entry point.            */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

NOTE The programmer should ensure that the memory ranges he specifies in the `SECTIONS` block are valid addresses for the controller he is using. In addition, when using the SDI debugger the addresses specified for code or constant sections must be located in the target board ROM area. Otherwise, the debugger will not be able to load the application

How to initialize the Vector table

The vector table can be initialized in the assembly source file or in the linker parameter file. We recommend that you initialize it in the linker parameter file.

- on page 403Initializing the Vector table in the linker PRM file on page 403 (recommended),
- Initializing the Vector Table in a source file using a relocatable section on page 405, or
- Initializing the Vector Table in a source file using an absolute section on page 408.

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 13.10 on page 403). All these labels must be published, otherwise they cannot be addressed in the linker `PRM` file.

Listing 13.10 Initializing the Vector table from a PRM File

```
          XDEF  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 on occurrence of an interrupt. 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 13.11 on page 404).

Listing 13.11 Using the VECTOR ADDRESS Linker Command

```
LINK test.abs
NAMES
  test.o
END
SECTIONS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0B00 TO 0x0CFF;
  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 13.12 on page 405.

Listing 13.12 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 13.13 on page 407).

Listing 13.13 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 13.14 on page 408 for an example.

Listing 13.14 Initializing the Vector Table using an absolute section

```
         XDEF ResetFunc
DataSec: SECTION
Data:    DS.W  5  ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
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 13.15 on page 409).

Listing 13.15 Example linker parameter file for Listing 13.14 on page 408:

```
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 different modules

Complex application or application involving several programmers can be split into several simple modules. In order to avoid any problem when merging the different modules, 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 13.16 on page 410 for an example `Test1.asm` include file.

Listing 13.16 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 13.17 on page 411 for an example `Test1.inc` include file.

Listing 13.17 Separating Code into Modules — Test1.inc

```
         XREF AddSource
; The AddSource function adds the value stored in the variable
; Source to the contents of the A register. The result of the
; computation is stored in the Source variable.
;
; Input Parameter:  The A register contains the value that should be
;                   added to the Source variable.
; Output Parameter: Source contains the result of the addition.

         XREF Source
; The Source variable is a 1-byte variable.
```

Example of an assembly File (Test2.asm)

Listing 13.18 on page 411 is another assembly code file module for this project.

Listing 13.18 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 13.19 Separating assembly code into modules—Test2.prm

```
LINK test2.abs /* Name of the executable file generated. */
NAMES
  test1.o
  test2.o / *Name of the object files building the application. */
END

SECTIONS
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF; /* READ_ONLY  mem. */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF; /* READ_WRITE mem. */
END

PLACEMENT
  /* variables are allocated in MY_RAM          */
  DataSec, DEFAULT_RAM           INTO MY_RAM;

  /* code and constants are allocated in MY_ROM */
  CodeSec, ConstSec, DEFAULT_ROM INTO MY_ROM;
END
INIT  entry      /* Definition of the application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Definition of the reset vector. */
```

NOTE The `'CodeSec'` section is defined in both object files. In `'test1.o'`, the `'CodeSec'` section contains the symbol `'AddSource'`. In `'test2.o'`, the `'CodeSec'` section contains the `'entry'` symbol. According to the order in which the object files are listed in the `NAMES` block, the function `'AddSource'` is allocated first and the `'entry'` symbol is allocated next to it.

Using the direct addressing mode to access symbols

There are different ways for the Assembler to use the direct addressing mode on a symbol:

- "Using the direct addressing mode to access external symbols" on page 413,
- "Using the direct addressing mode to access exported symbols" on page 413,
- "Defining symbols in the direct page" on page 413,
- "Using the force operator" on page 414, or

- “Using SHORT sections” on page 414.

Using the direct addressing mode to access external symbols

External symbols, which should be accessed using the direct addressing mode, must be declared using the `XREF.B` directive. Symbols which are imported using `XREF` are accessed using the extended addressing mode.

Listing 13.20 Using direct addressing to access external symbols

```
      XREF.B ExternalDirLabel
      XREF   ExternalExtLabel
…
      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 13.21 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 13.22 on page 414).

Listing 13.22 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 13.23 on page 414).

The supported force operators are:

- < or .B to force direct addressing mode
- > or .W to force extended addressing mode.

Listing 13.23 Using a force operator

```
...
dataSec: SECTION
label:   DS.B  5
...
codeSec: SECTION
...
         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 13.24 on page 415).

Listing 13.24 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” on page 419
- “Local Configuration File Entries” on page 429
- “MASM Compatibility” on page 441
- “MCUasm Compatibility” on page 445

A

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

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 A.1 on page 427 shows a typical `mcutools.ini` file.

Listing A.1 Typical mcutools.ini file layout

```
[Installation]
Path=c:\Freescale
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\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
```

B

Local Configuration File Entries

This appendix documents the sections and entries that can appear in the local configuration file. Usually, you name this file `project.ini`, where `project` is a placeholder for the name of your project.

A `project.ini` file can contains these sections:

- [Editor] Section on page 430
- [XXX_Assembler] Section on page 432
- Example on page 439

[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 statusbar state.

- 1: Statusbar is visible
- 0: Statusbar is hidden

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

ToolbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Description

Current toolbar state:

- 1: Toolbar is visible
- 0: Toolbar is hidden

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

WindowPos

Arguments

10 integers, e.g., “`0,1,-1,-1,-1,-1,390,107,1103,643`”

Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Changes of this entry do not show the `"*"` in the title.

Description

This numbers contain the position and the state of the window (maximized,..) and other flags.

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

WindowFont

Arguments

`size:` = 0 -> generic size, < 0 -> font character height, > 0 -> font cell height

`weight`: 400 = normal, 700 = bold (valid values are 0..1000)

`italic`: 0 = no, 1 = yes

`font name`: max. 32 characters.

Description

Font attributes.

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

Example

```
WindowFont=-16,500,0,Courier
```

TipFilePos

Arguments

any integer, e.g., 236

Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

Saved

Always when saving a configuration file.

ShowTipOfDay

Arguments

0/1

Description

Should the *Tip of the Day* dialog box be shown at startup?

- 1: It should be shown
- 0: No, only when opened in the help menu

Saved

Always when saving a configuration file.

Options

Arguments

current option string, e.g.: `-W2`

Description

The currently active option string. This entry can be very long.

Saved

Only with *Options* set in the *File* > *Configuration* > *Save Configuration* dialog box.

EditorType

Arguments

0/1/2/3/4

Description

This entry specifies which editor configuration is active:

- 0: global editor configuration (in the file `mcutools.ini`)
- 1: local editor configuration (the one in this file)
- 2: command line editor configuration, entry EditorCommandLine
- 3: DDE editor configuration, entries beginning with EditorDDE
- 4: CodeWarrior with COM. There are no additional entries.

For details, see also Editor Setting dialog box.

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

EditorCommandLine

Arguments

Command line, for `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 B.1 on page 439 shows a typical layout of the configuration file (usually `project.ini`).

Listing B.1 Example of a project.ini file

```
[Editor]
Editor_Name=IDF
Editor_Exe=c:\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
```

C

MASM Compatibility

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

Comment Line

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

Constants (Integers)

For compatibility with the MASM Assembler, the following notations are also supported for integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9) followed by a 'd' or 'D' character.
- A hexadecimal constant is defined by a sequence of hexadecimal digits (0-9, a-f, A-F) followed by a 'h' or 'H' character.
- An octal constant is defined by a sequence of octal digits (0-7) followed by an 'o', 'O', 'q', or 'Q' character.
- A binary constant is defined by a sequence of binary digits (0-1) followed by a 'b' or 'B' character.

Listing C.1 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 C.1 on page 442 are also supported for operators:

Table C.1 Operator notation for MASM compatibility

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

Directives

Table C.2 on page 442 enumerates the directives that are supported by the Macro Assembler for compatibility with MASM:

Table C.2 Supported MASM directives

Operator	Notation	Description
RMB	DS	Define storage for a variable. Argument specifies the byte size
RMD	DS 2*	Define storage for a variable. Argument specifies the number of 2-byte blocks
RMQ	DS 4*	Define storage for a variable. Argument specifies the number of 4-byte blocks
ELSEC	ELSE	Alternate of conditional block
ENDC	ENDIF	End of conditional block
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user defined title for the assembler listing file.

Table C.2 Supported MASM directives

Operator	Notation	Description
GLOBAL	XDEF	Make a symbol public (Visible from outside)
PUBLIC	XDEF	Make a symbol public (Visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.
XREFB	XREF.B	Import reference to an external symbol located on the direct page.
SWITCH		Allows the switching to a section which has been defined previously.
ASCT		Creates a predefined section which name id ASCT.
BSCT		Creates a predefined section which name id BSCT. Variable defined in this section are accessed using the direct addressing mode.
CSCT		Creates a predefined section which name id CSCT.
DSCT		Creates a predefined section which name id DSCT.
IDSCT		Creates a predefined section which name id IDSCT.
IPSCT		Creates a predefined section which name id IPSCT.
PSCT		Creates a predefined section which name id PSCT.

D

MCUasm Compatibility

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

MCUasm compatibility mode can be activated, specifying the `-MCUasm` option.

This chapter covres the following topics:

Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even when they start on column 1.

When MCUasm compatibility mode is activated, following portion of code generate an error message, because the label 'label' is not followed by a colon.

Listing D.1 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 D.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 absolute expressions.

Obsolete directives

Table D.1 on page 446 enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.:

Table D.1 Obsolete directives

Operator	Notation	Description
RMB	DS	Define storage for a variable
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (Visible from the outside)
PUBLIC	XDEF	Make a symbol public (Visible from the outside)
EXTERNAL	XREF	Import reference to an external symbol.

Index