S12(X) Assembler Manual

Revised: 27 February 2009

How to Contact Us

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

Table of Contents

I Using the S12(X) Assembler

I

Using the S12(X) Assembler

This document explains how to effectively use the S12(X) Macro Assembler.

Highlights

The major features of the S12(X) Assembler are:

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

Structure of this Document

This book includes the following chapters:

- Working with the Assembler: A tutorial for creating assembly-language projects using the CodeWarrior™ Development Suite or the standalone build tools. Both relocatable and absolute assembly projects are created. Also a description of the Assembler's environment that creates and edits assembly source code and assembles the source code into object code which could be further processed by the Linker.
- Assembler Graphical User Interface: A description of the Macro Assembler's Graphical User Interface (GUI)
- Environment: A detailed description of the Environment variables used by the Macro Assembler
- Files: A description of the input and output file the Assembles uses or generates.
- Assembler Options: A detailed description of the full set of assembler options
- Sections: A description of the attributes and types of sections

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

The Appendices chapters include:

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

1

Working with the Assembler

This chapter is primarily a tutorial for creating and managing S12(X) assembly projects with the CodeWarrior™ Development Studio. In addition, there are instructions to utilize the Assembler and Smart Linker build tools in the CodeWarrior Development Studio for assembling and linking assembly projects.

This chapter covers the following topics:

Programming Overview

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

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

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

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

This manual covers the Assembler designed for the Freescale 16-bit S12(X) series of microcontrollers. There is a companion manual for this series that covers the S12(X) Compiler.

The S12(X) Assembler can be used as a transparent, integral part of the CodeWarrior Development Studio. This is the recommended way to get your project up and running in minimal time. Alternatively, the Assembler can also be configured and used as a standalone macro assembler as one of the Build Tool Utilities included with the CodeWarrior Development Studio, such as a Linker, Compiler, ROM Burner, Simulator or Debugger.

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

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

Project Directory

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

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

External Editor

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

Using CodeWarrior Development Studio to Manage Assembly Language Project

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

This section will create a basic CodeWarrior project that uses S12(X) assembly source code exclusively - no C source code. A sample program is included for a project created using the Wizard. For example, the program included for an assembly project calculates the next number in a mathematical Fibonacci series. It is much easier to analyze any program if you already have some familiarity with solving the result in advance. Therefore, the following paragraph describes a Fibonacci series.

A Fibonacci series is a mathematical infinite series that is very easy to visualize (Listing 1.1):

Listing 1.1 Fibonacci series

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

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

Let's now create a project with the CodeWarrior Development Studio and analyze the assembly source and the Linker's parameter files to calculate a Fibonacci series for a particular 16-bit microprocessor in the Freescale S12(X) family - in this case, the MC9S12DP512.

Using New Project Wizard to Create Project

This section demonstrates using the CodeWarrior IDE Wizard to create a new project.

1. Select **Start > Programs > Freescale CodeWarrior > CodeWarrior Development Studio for S12(X) V5.0 > CodeWarrior IDE**.

   The CodeWarrior IDE starts and the CodeWarrior Startup dialog box (Figure 1.1) appears.

NOTE If the CodeWarrior IDE application is already running, select **File > New Project**.

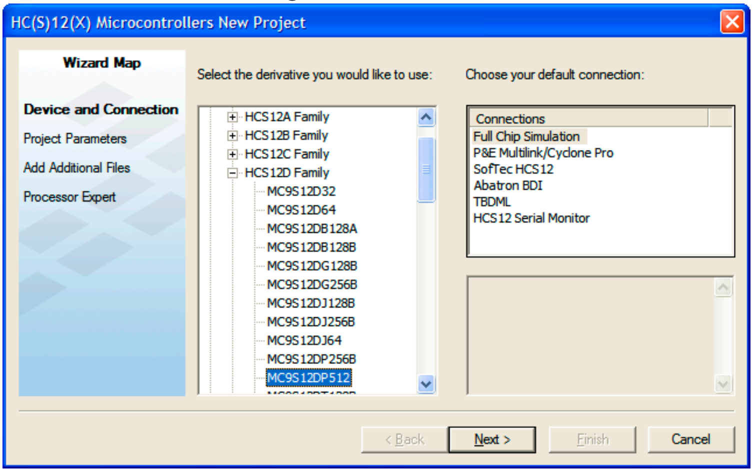
Figure 1.1 Startup Dialog Box

NOTE Here we use the MC9S12DP512 derivative and the Full Chip Simulation connection as an example.

2. Click the **Create New Project** button — the **Device and Connection** page (Figure 1.2) appears.

3. Select a derivative you would like to use. For example, select `HCS12 > HCS12D Family > MC9S12DP512`.

Figure 1.2 Device and Connection Page

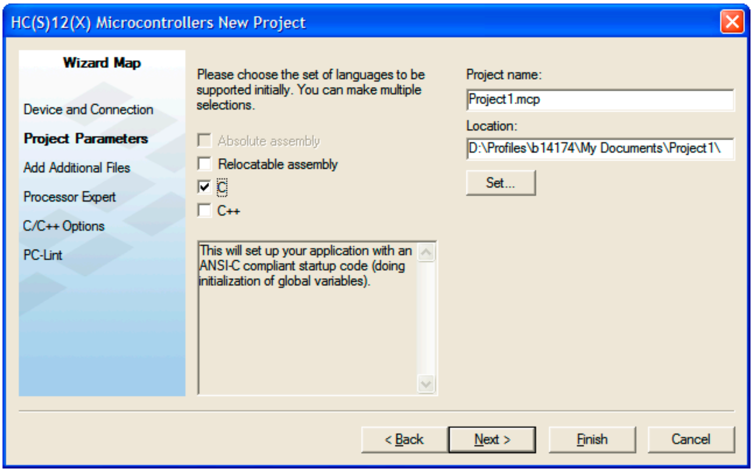


4. Select a default connection. For example, select `Full Chip Simulation`.
5. Click the **Next** button.

   The **Project Parameters** page (Figure 1.3) appears.

NOTE The wizard pages will change depending on the derivative selected in the **Device and Connections** page.

Figure 1.3 Project Parameters Page



6. Select the languages to be supported initially. For example, check the Relocatable Assembly checkbox. By default, the C checkbox is checked.
   - Absolute Assembly — Check to use only one single assembly source file with absolute assembly. There will be no support for relocatable assembly or linker. This option will appear grayed out if C is checked.
   - Relocatable Assembly — Check to split up the application into multiple assembly source files. The source files are linked together using the linker.

NOTE If an assembly project has two or more assembly source (`*.asm`) files, the *Relocatable Assembly* option must be selected. This is because it is more flexible to use relocatable assembly in case the project expands to include additional assembly source files. The *Absolute Assembly* option will be covered later in this chapter.

- C — Check to set your application with an ANSI-C compliant startup code (doing initialization of global variables).

NOTE Clearing the C checkbox will remove the C/C++ Options and PC-Lint pages from the Wizard map.

- C++ — Check to set your application with an ANSI-C compliant startup code (doing initialization of global variables).

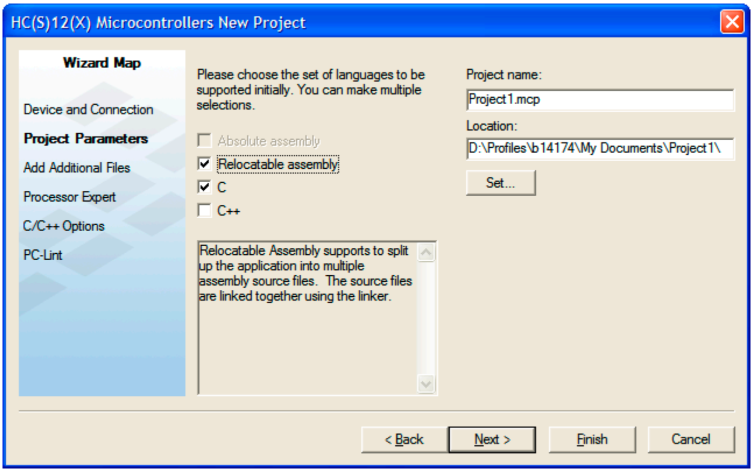
7. In the Project name text box, specify the name of the new project. For example, Project1.

8. To specify a location other than the default location, enter the new path in the Location text box or click the **Set** button to browse the folder location.

NOTE The IDE automatically creates a folder with the same name in specified location. The IDE automatically adds `.mcp` extension when it creates project.

NOTE You can select the **Finish** button to accept defaults for remaining options.

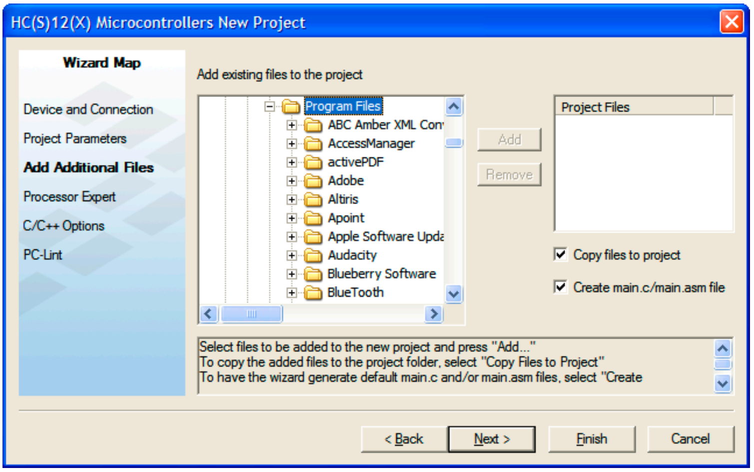
Figure 1.4 Project Parameters Page - Relocatable Assembly



9. Click the **Next** button.

   The **Add Additional Files** page (Figure 1.5) appears.

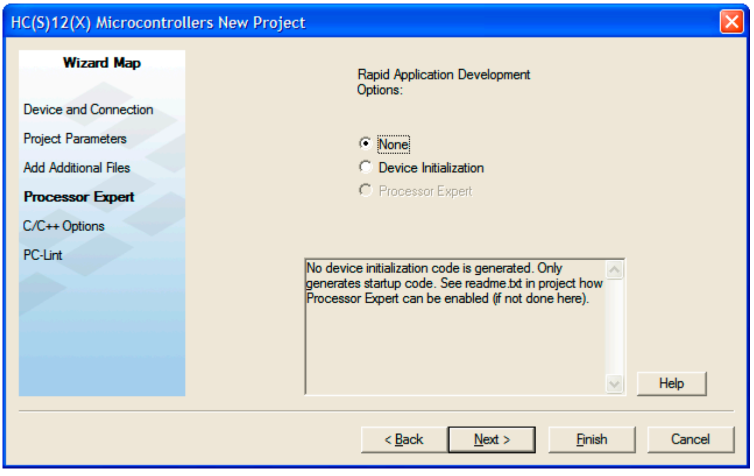
Figure 1.5 Add Additional Files Page



10. Select the files to be added to the project and click the **Next** button.

    The **Processor Expert** page (Figure 1.6) appears.

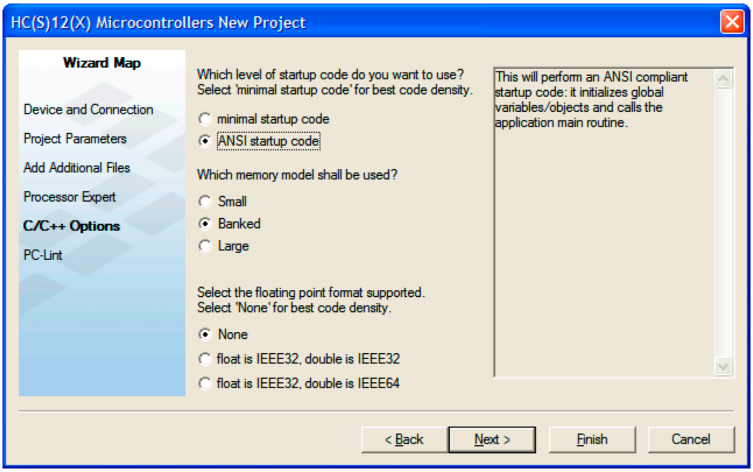
Figure 1.6 Processor Expert Page



11. Select the None option button if you do not want to generate the device initilization code.
12. Click the **Next** button.

    The **C/C++ Options** page (Figure 1.7) appears.

Figure 1.7 C/C++ Options Page



13. Select the minimal startup code for best code density. By default, the *ANSI startup code* option button is selected.

    - Minimal startup code — Initializes the stack pointer and calls the main function. No initialization of global variables is done, giving the user the best speed/code density and a fast startup time. But, the application code has to care about variable initialization. This makes this option not ANSI compliant, since ANSI requires variable initialization.
    - ANSI startup code — Performs an ANSI-compliant startup code that initializes global variables/objects and calls the application main routine.

14. Select the memory model to use. By default, the *Small* memory model option button is selected.

    - Small — Use if both the code and the data fit into the 64kB address space. By default all variables and functions are accessed with 16-bit addresses. the compiler does support banked functions or paged variables in this memory model, but all accesses have to be explicitly handled.
    - Banked — Uses banked function calls by default. The default data access however is still 16 bit. Because the overhead of the far function call is not very large, this memory model suits all applications with more than 64k code. Data paging can be used in the banked memory model, however all far objects and pointers to them have to be specially declared.
    - Large — Use to support upports both code banking and data paging by default. But especially the data paging does cause a great overhead and therefore should only be used with care. The overhead is significant with respect to both code size and speed.

If it is possible to manually use far accesses to the data, which does not fit into the 64 bit address space, then the banked memory model should be used instead.

15. Select the floating point format support. By default, the *None* option button is selected.
    - None — Use if you do not want to use floating point for the project.
    - Float is IEEE32, double is IEEE32 — All float and double variables are 32-bit IEEE32 for the HC08.
    - Float is IEEE32, double is IEEE64 — Float variables are 32-bit IEEE32. Double variables are 64-bit IEEE64 for the project.

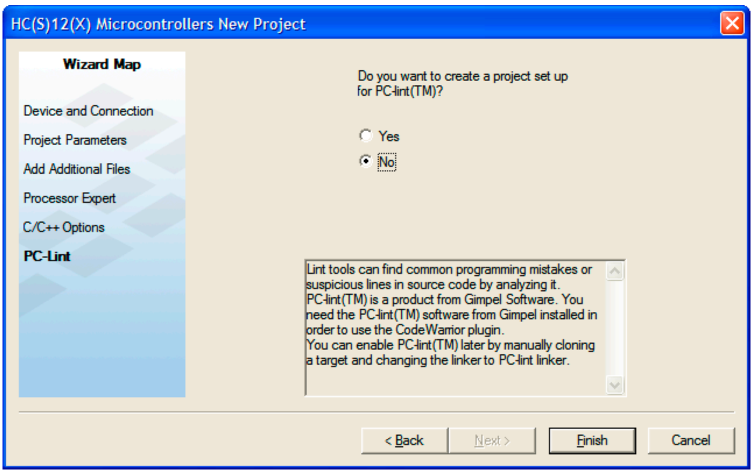
NOTE These three are the default selections and are the routine entries for an ANSI-C project. Floating point numbers impose a severe speed-hit penalty, so use the integer number format whenever possible.

NOTE If you intend to use the flexible type management option (-T), choose minimal startup code instead of ANSI startup code. The ANSI C-compliant startup code does not support 8-bit `int`.

16. Click the **Next** button.

    The **PC-Lint** page (Figure 1.8) appears.

Figure 1.8 PC-Lint Page

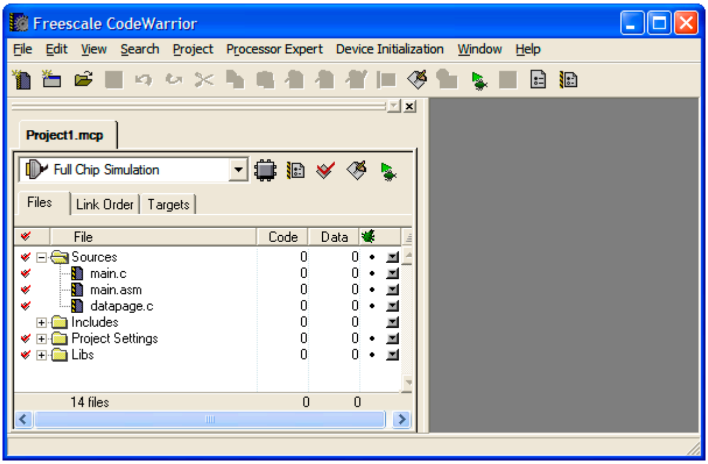


17. Select the *Yes* option button if you want to create a project set up for PC-Lint. By default, the *No* option button is selected.

18. Click the **Finish** button.

    IDE creates a project according to your specifications and the **Project** window (Figure 1.9) appears, docked at the left side of the main window.

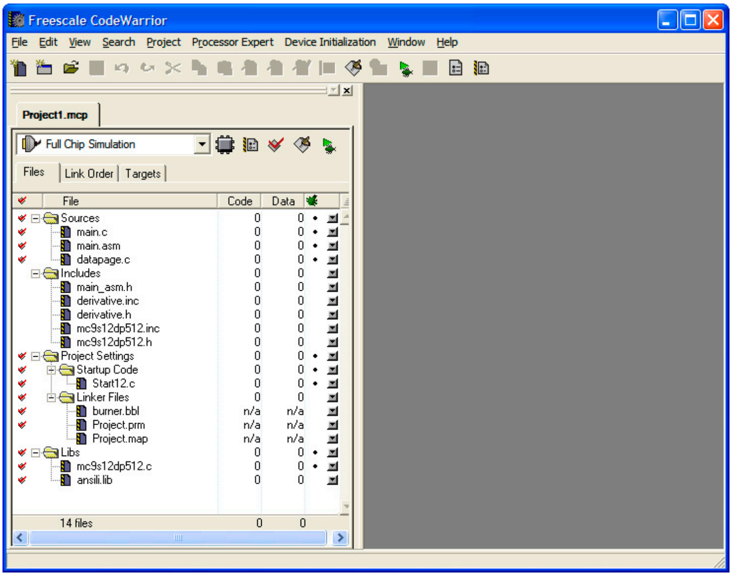
Figure 1.9 Project Window



NOTE You can (but do it later) safely close the CodeWarrior IDE at any time after this point, and your project will be automatically configured in its previously-saved status when you work on the project later. Using the New Project wizard, an S12(X) 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. .

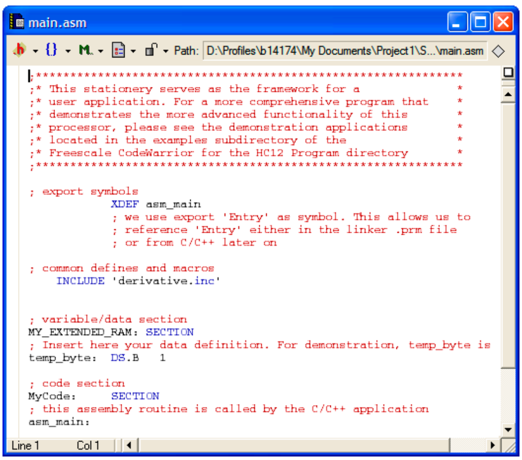
If you expand the folder icons (actually groups of files), by clicking in the CodeWarrior project window, you could view the files that the CodeWarrior Assembler generated. In general, any folders or files in the project window with red check marks will remain so checked until the files are successfully assembled, compiled, or linked (Figure 1.10).

Figure 1.10 Expanded View of Folders



19. Double-click the `main.asm` file in the **Sources** group. The CodeWarrior editor opens the `main.asm` file (Figure 1.11).

Figure 1.11 `main.asm` File in the Editor Window



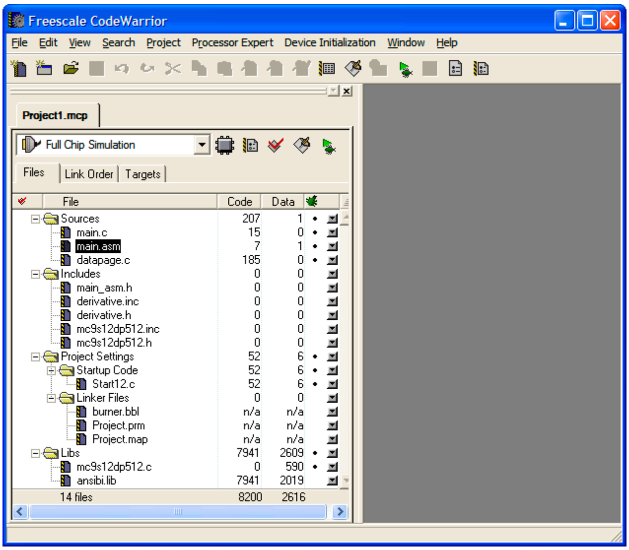
You can use this default `main.asm` file as a base to later rewrite your own assembly source program. Otherwise, you can import other assembly-code files into the project and instead delete the default `main.asm` file from the project. For this project, the `main.asm` file contains the sample Fibonacci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

As a precaution, you can determine if the project is configured correctly and the source code is free of syntactical errors. It is not necessary that you do so, but you should make (build) the default project that the CodeWarrior Assembler just created.

20. Select **Project > Make** to build and link code in the application. Alternatively, you can click the  icon on the project window the toolbar.

NOTE All the red check marks will disappear if the project build is successful (Figure 1.12).

Figure 1.12 Project Window After Successful Build

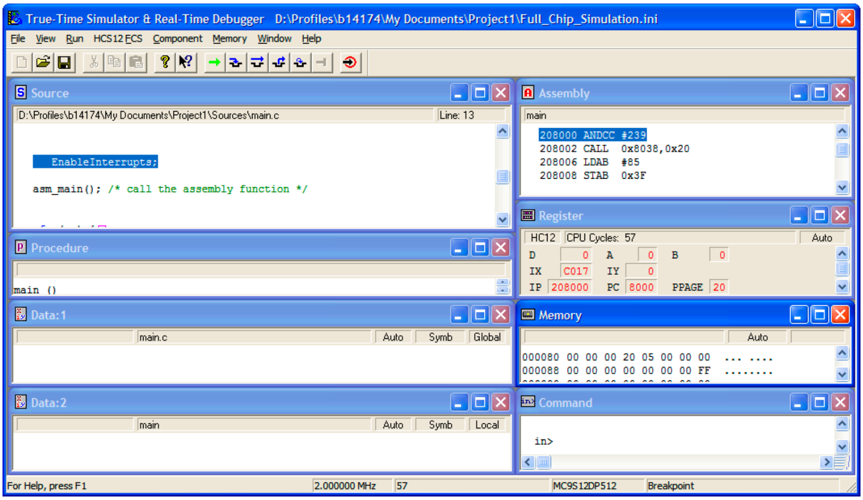


NOTE The Code and Data columns in the project window show that the code size is 8200 bytes and the data size is 2616 bytes after assembling the `main.asm` file.

21. Select **Project > Debug** to start the debugger. Alternatively, you can click the  icon.

The**True-Time Simulator & Real-Time Debugger** window opens (Figure 1.13).

Figure 1.13 True-Time Simulator & Real-Time Debugger Window



Analysis of Groups and Files in Project Window

There are three default groups and one default subgroup for holding project’s files. It really does not matter in which group a file resides as long as that file is somewhere in the project window. A file does not even have to be in any group. The groups do not correspond to any physical folder in the project directory. They are simply present in the project window for conveniently grouping files anyway you choose. You can add, rename, or delete files or groups, or 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 project has an include file for the particular CPU derivative. In this case, the group contains the `mc9s12a512.inc` file for the MC9S12XDP512 derivative.

- Project Settings

  This groups includes the Startup Code and Linker Files groups.

– Startup Code

  This group contains `Start12.c` file.

  Linker Files
  This group contains the `burner.bbl`, `Project.prm`, and *`Project Name.map`* files.

- Libs
  - This group contains the `mc9s12dp512.c` and `ansili.lib` file

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

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

Writing Assembly Source Files

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

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

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

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

Analyzing Project Files

We will analyze the default `main.asm` file located in the `Sources` folder created by the New Project Wizard. Listing 1.2 illustrates the default `main.asm` file.

Listing 1.2 `main.asm` file

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

; common defines and macros
    INCLUDE 'derivative.inc'



; variable/data section
MY_EXTENDED_RAM: SECTION
                               ; Insert here your data definition. For
                               ; demonstration,temp_byte is used.
temp_byte:    DS.B    1
```

```
; code section
MyCode:     SECTION
; this assembly routine is called by the C/C++ application
asm_main:
                   MOVB    #1,temp_byte        ; just some
                                               ;demonstration code
                   ;NOP                        ; Insert here your own
                                               ;code
                   ;RTC                        ;return to caller
```

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

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

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

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

Assembling Source Files

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

Assembling with CodeWarrior

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

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

The object files are generated and placed into the `ObjectCode` subfolder in the project directory. 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\<target_name>\ObjectCode\main.asm.o.

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

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

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

- Project.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.

- Project.abs.s19

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

TIP The remaining file in the default `bin` subfolder is the `main.dbg` file that was generated back when the `main.asm` file was successfully assembled. This debugging file was generated because a bullet was present in the debugging column in the project window.
You can enter (or deselect by subsequently toggling) a debugging bullet by clicking at the intersection of the `main.asm` file (or whatever other source code

file selected for debugging) and the debugging column in the project window. Whenever the Debugger or Simulator does not show a desired file in its `Source` window, check first to see if the debugging bullet is present or not in the project window. The bullet must be present for debugging purposes.

TIP The New Project Wizard does not generate default assembler-output listing files. If you want such listing files generated, you have to select this option: **Edit >** <*target_name*> **Settings > Target > Assembler for HC12 > Options.** Select the **Output** tab in the **HC12 Assembler Option Settings** dialog box. Check *Generate a listing file* and *Do not print included files in listing file*. (You can uncheck *Do not print included files in listing file* if you choose, but be advised that the include files for CPU derivatives are usually quite lengthy.) Now a `*.lst` file will be generated or updated in the `bin` subfolder of the project directory whenever a `*.asm` file is assembled.

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

Assembling with Standalone Assembler

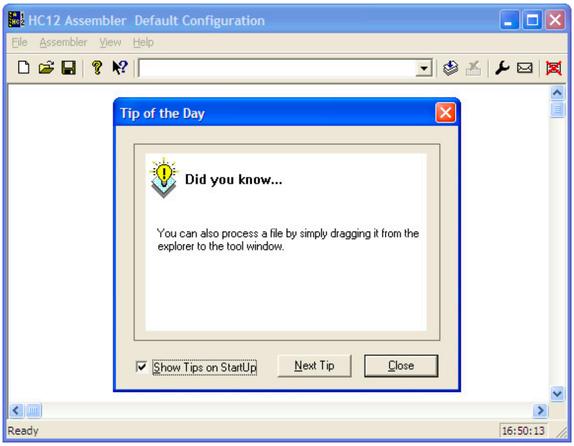
It is also possible to use the S12(X) Assembler as a standalone assembler. (If you already have an assembled source file and prefer not to use the Assembler but do want to use the Linker, you can skip this section and proceed to “Linking Application” on page 48.)

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

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

Start the Assembler by opening the `ahc12.exe` file located in `<CodeWarrior Install dir>\Prog` folder. The Assembler opens (Figure 1.14).

Figure 1.14 HC12 Assembler Default Configuration Window



Read any of the Tips if you want to and then click the **Close** button to close the **Tip of the Day** dialog box.

NOTE If you do not want to display Tips on startup, clear the Show Tips on StartUp checkbox.

Configuring Assembler

The 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, such as listings of the most recent projects. All tools may store some global data in the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (e.g. `C:\WINDOWS`). See Listing 1.3.

Listing 1.3 Typical locations for a global configuration file

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

If a tool is started in the `C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\prog\` directory, the initialization file in the same directory as the tool is used.

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

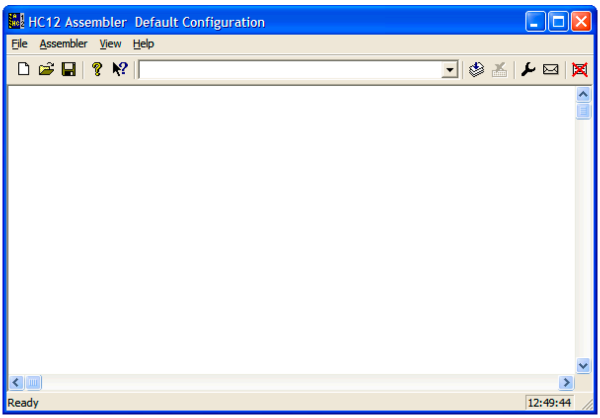
  This file could be used by any Build Tool for a particular project. For information about entries for the local configuration file, see Local Configuration File Entries in the Appendices.

After opening the assembler, you would load the configuration file for your project if it already had one. In this case, you will create a new configuration file and save it so that whenever the project is reopened, its previously saved configuration state will be used.

1. From the **File** menu, select **New / Default Configuration**.

   The **HC12 Assembler Default Configuration** window appears (Figure 1.15)

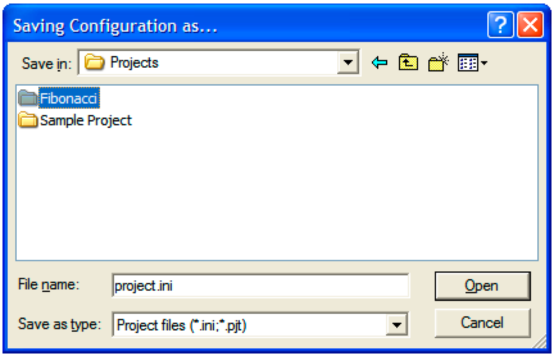
Figure 1.15 HC12 Assembler Default Configuration Window



2. Save this configuration in a newly created folder that will become the project directory.
3. Select **File** > **Save Configuration**.

   A **Save Configuration as** dialog box appears. Navigate to the folder of your choice and create and name a folder and filename for the configuration file (Figure 1.16).

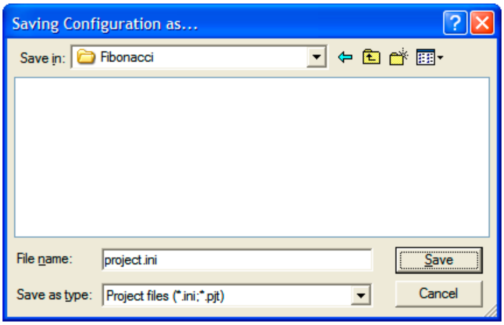
Figure 1.16 Saving Configuration as... Dialog Box



4. Click the **Open** button.

   The current directory for the assembler changes to your project directory (Figure 1.17).

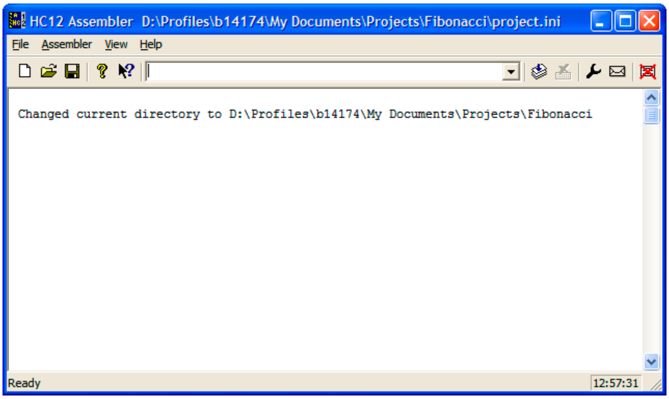
Figure 1.17 Saving Configuration As



5. Click the **Open** button.

   The current directory for the assembler changes to your project directory (Figure 1.18).

Figure 1.18 Assembler Changed Current Directory to Project Directory



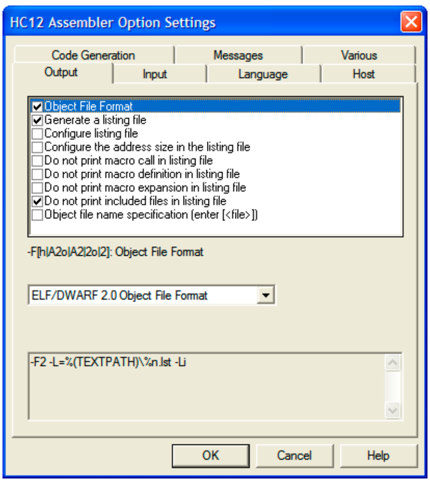
If you were to examine the project directory with the Windows Explorer at this point, it would only contain the *<project_name>.ini* configuration file that you just created. Any options added to or deleted from your project by any Build Tool would be placed into or deleted from this configuration file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

You now set the object-file format that you intend to use (HIWARE or ELF/DWARF).

1. Select the menu entry **Assembler > Options**.

   The Assembler displays the **HC12 Assembler Option Settings** dialog box (Figure 1.19).

Figure 1.19 HC12 Assembler Option Settings Dialog Box



2. In the **Output** panel, select the checkboxes labeled **Generate a listing file** and **Object File Format**.
3. For the *Object File Format option*, select *ELF/DWARF 2.0 Object File Format*.

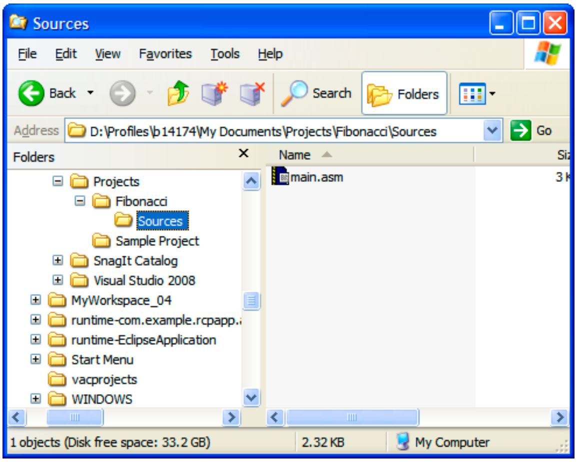
NOTE The listing file would be much shorter if the *Do not print included files in listing file* checkbox is checked, so you may want to select that option also.

4. In the **Code Generation** panel, select *Derivative.*
5. For the *Derivative option*, select *HCS12*.
6. Click the **OK** button to close the **HC12 Assembler Option Settings** dialog box.
7. Save the changes to the configuration by:
   - selecting **File > Save Configuration (Ctrl + S)** or
   - clicking the **Save** button on the toolbar.

Input Files

Now that the project's configuration is set, you can assemble an assembly-code file. However, the project does not contain any source-code files at this point. You could create assembly `*.asm` and include `*.inc` files from scratch for this project. However, for simplicity's sake, you can copy and paste the `Sources` folder from a previous CodeWarrior project into the project directory (Figure 1.20).

Figure 1.20 Project Files



Now there are two files in the project:

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

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

Listing 1.4 `main.asm` file

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

; export symbols
            XDEF asm_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.


; common defines and macros
            INCLUDE 'derivative.inc'

; variable/data section
MY_EXTENDED_RAM: SECTION
                              ;Insert here your data definition here.
                              ;For demonstration, temp_byte is used.
temp_byte:  DS.B 1

; code section
MyCode:     SECTION
           ; this assembly routine is called by the C/C++ application
asm_main:
MOVB   #1,temp_byte   ; just some demonstration code
            NOP                        ; Insert here your own code

            RTC                        ; return to caller
```

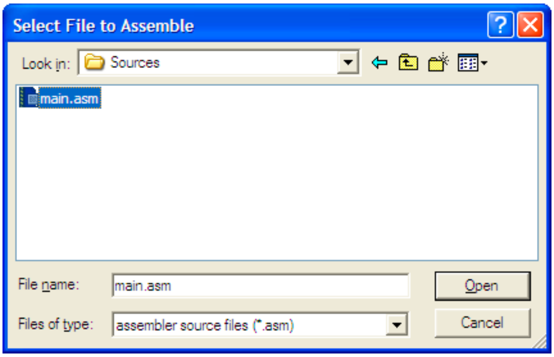
Assemblingn Assembly Source-Code Files

To assemble the `main.asm` file, perform these steps:

1. Select **File > Assemble**.

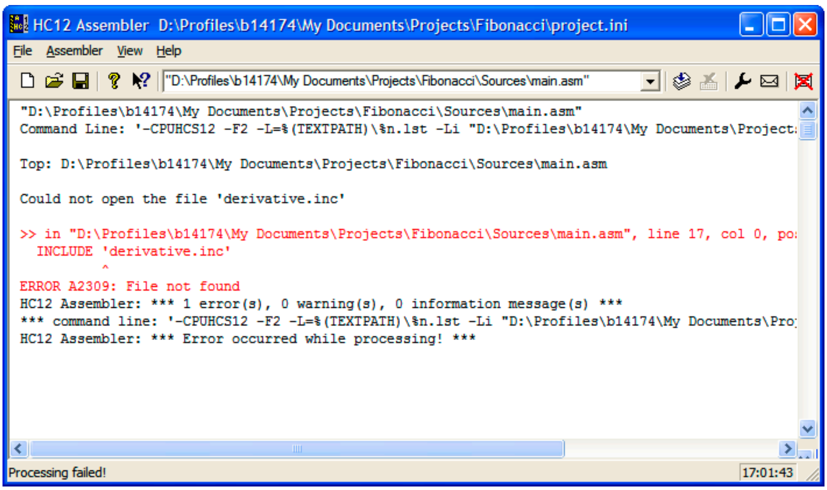
   The **Select File to Assemble** dialog box appears (Figure 1.21).

Figure 1.21 Select File to Assemble Dialog Box



2. Browse to the *`Sources`* folder in the project directory and select the *`main.asm`* file.
3. Click the **Open** button and the `main.asm` file should start assembling (Figure 1.22).

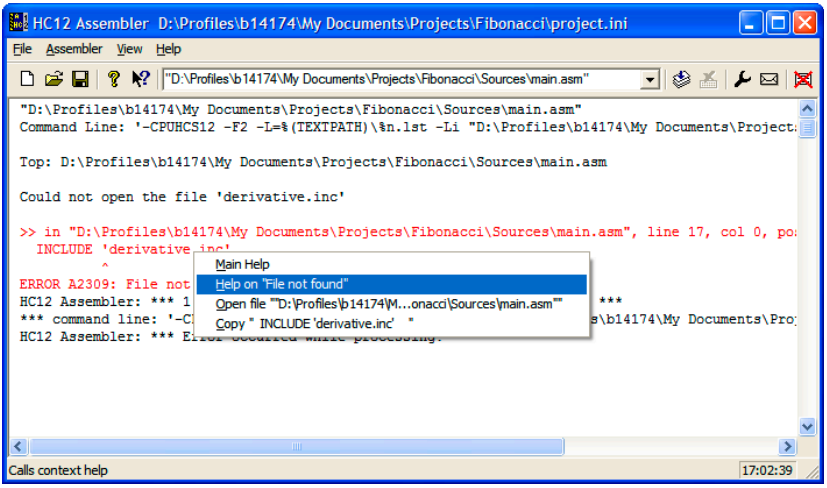
Figure 1.22 Results of Assembling the `main.asm` File



The project window provides positive information about the assembly process or generates error messages if the assembly was unsuccessful. In this case an error message is generated. - the *A2309 File not found* message.

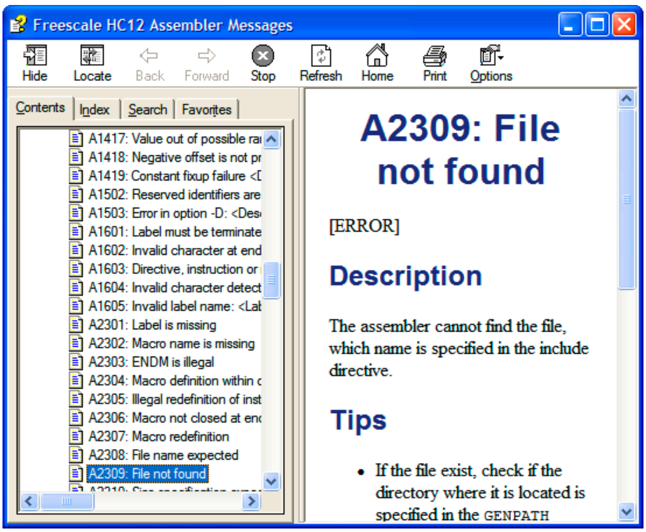
4. Right-click on the text about the error message, a context menu appears (Figure 1.23).

Figure 1.23 Context Menu



5. Select the *Help on "file not found"* option and help for the A2309 error message appears (Figure 1.24).

Figure 1.24 A2309 Error Message Help



The help message for the A2309 error states that the Assembler looks for this "missing" include file first in the current directory and then in the directory specified by the `GENPATH` environment variable. This implies that the `GENPATH` environment variable should specify the location of the `derivative.inc` include file.

NOTE If you read the `main.asm` file, you could have anticipated this on account of this statement on line 10: INCLUDE 'derivative.inc'.

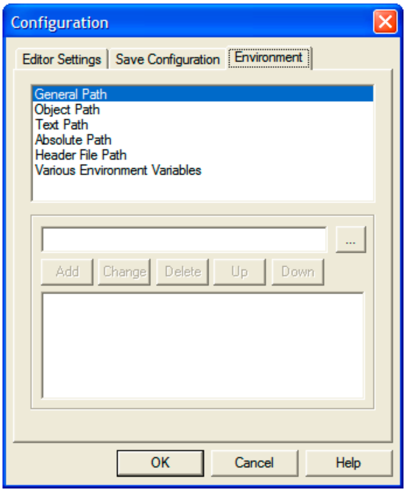
To fix this, perform these steps:

1. Select **File > Configuration**.

   The **Configuration** dialog box appears (Figure 1.25).

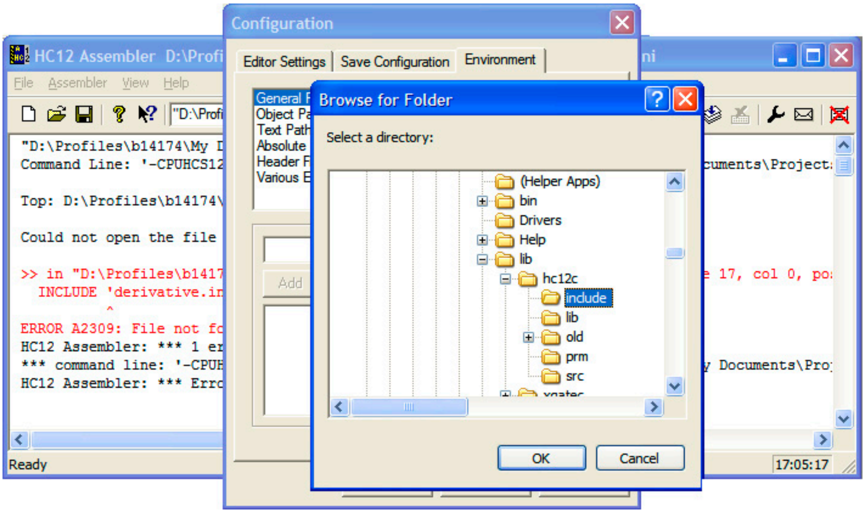
2. Select the **Environment** tab and then select **General Path**.

Figure 1.25 Configuration Dialog Box



3. Click the "**...**" button and navigate in the **Browse for Folder** dialog box for the folder that contains the missing file - the `include` subfolder in the CodeWarrior installation's `lib` folder.

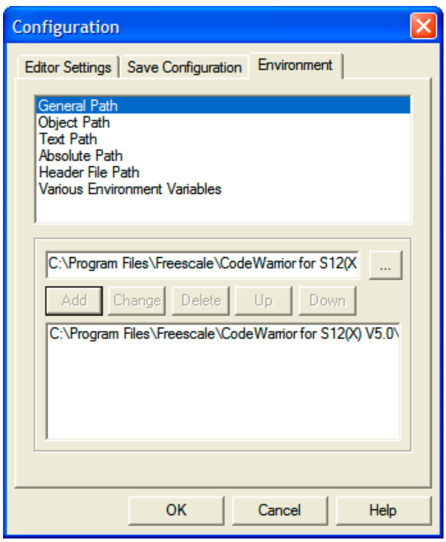
Figure 1.26 Browsing for Sources Folder



4. Click the **OK** button to close the **Browse for Folder** dialog box.

   The **Configuration** dialog box is now active (Figure 1.27).

Figure 1.27 Adding GENPATH



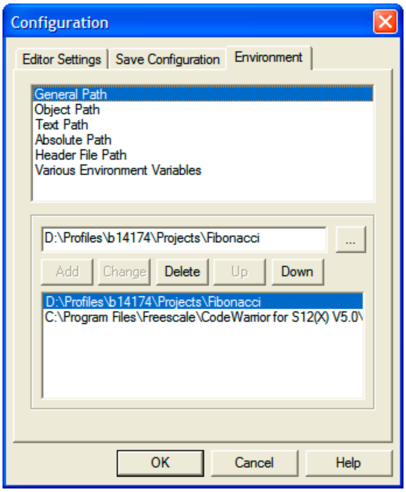
5. Click the **Add** button, and the path to "*<CodeWarrior Installation>\lib\hc12c\include*" now appears in the lower panel.
6. Click the **OK** button. An asterisk appears in the *Title bar*, so save the change to the configuration by clicking the **Save** button or by selecting **File > Save Configuration**. The asterisk disappears when the file is saved.

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

---

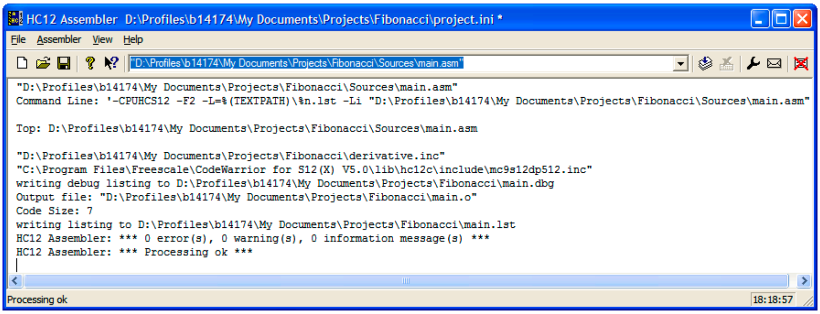
7. After the `GENPATH` is set up for the include file, copy the `derivative.inc` file from an existing project to the recently created *`Sources`* folder.
8. Set the `GENPATH` for the `derivative.inc` file (Figure 1.29).

Figure 1.28 Adding GENPATH — derivative.inc



9. Click the OK button to close the **Configuration** dialog box.
10. Select **File > Assemble,** navigate to the *`main.asm`* file and click the **Open** button (Figure 1.29).

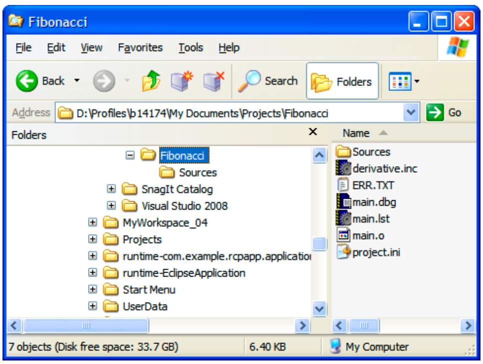
Figure 1.29 Successful Assembly - `main.o` Object File Created



The Macro Assembler indicates successful assembling and indicated that the code size is 7 bytes. The message "`*** 0 error(s),`" indicates that the `main.asm` file assembled without errors. Do not forget to save the configuration one additional time.

The Macro Assembler generated a `main.dbg` file (for use with the simulator/debugger), a `main.o` object file (for further processing with the Linker), and a `main.lst` output listing file in the project directory. The binary object file has the same name as the input module, but with the *.o extension - `main.o`. The debug file has the same name as the input module, but with the `*.dbg` extension - `main.dbg`. The assembly output file is similarly named - `main.lst`. The `ERR.TXT` file was generated as a result of the first failed attempt to assemble the `main.asm` file without the correct path to the *.inc file (Figure 1.30).

Figure 1.30 Project Directory After a Successful Assembly



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

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

Linking Application

Once the object files are available you can link your application. The linker organizes the code and data sections into ROM and RAM memory areas according to the project's linker parameter (PRM) file. The Linker's input files are object-code files from the assembler or compiler, library files, and the Linker PRM file.

Linking with CodeWarrior

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

Listing 1.5 Linker PRM file for the MC9S12DP512 derivative

```
//* This is a linker parameter file for the MC9S12DP512 */
NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */

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

/* Register space  */
/*    IO_SEG         = PAGED         0x0000 TO   0x03FF; intentionally
not defined */

/* EPROM */
      EEPROM         = READ_ONLY     0x0400 TO   0x07FF;

/* RAM */
      RAM            = READ_WRITE    0x0800 TO   0x3FFF;

/* non-paged FLASHs */
      ROM_4000       = READ_ONLY     0x4000 TO   0x7FFF;
      ROM_C000       = READ_ONLY     0xC000 TO   0xFEFF;
 /*   VECTORS        = READ_ONLY     0xFF00 TO   0xFFFF; intentionally
not defined: used for VECTOR commands below */
   //OSVECTORS       = READ_ONLY     0xFF8C TO   0xFFFF;   /* OSEK
interrupt vectors (use your vector.o) */
```

```
/* paged FLASH:                    0x8000 TO   0xBFFF; addressed
through PPAGE */
      PAGE_20       = READ_ONLY   0x208000 TO 0x20BFFF;
      PAGE_21       = READ_ONLY   0x218000 TO 0x21BFFF;
      PAGE_22       = READ_ONLY   0x228000 TO 0x22BFFF;
      PAGE_23       = READ_ONLY   0x238000 TO 0x23BFFF;
      PAGE_24       = READ_ONLY   0x248000 TO 0x24BFFF;
      PAGE_25       = READ_ONLY   0x258000 TO 0x25BFFF;
      PAGE_26       = READ_ONLY   0x268000 TO 0x26BFFF;
      PAGE_27       = READ_ONLY   0x278000 TO 0x27BFFF;
      PAGE_28       = READ_ONLY   0x288000 TO 0x28BFFF;
      PAGE_29       = READ_ONLY   0x298000 TO 0x29BFFF;
      PAGE_2A       = READ_ONLY   0x2A8000 TO 0x2ABFFF;
      PAGE_2B       = READ_ONLY   0x2B8000 TO 0x2BBFFF;
      PAGE_2C       = READ_ONLY   0x2C8000 TO 0x2CBFFF;
      PAGE_2D       = READ_ONLY   0x2D8000 TO 0x2DBFFF;
      PAGE_2E       = READ_ONLY   0x2E8000 TO 0x2EBFFF;
      PAGE_2F       = READ_ONLY   0x2F8000 TO 0x2FBFFF;
      PAGE_30       = READ_ONLY   0x308000 TO 0x30BFFF;
      PAGE_31       = READ_ONLY   0x318000 TO 0x31BFFF;
      PAGE_32       = READ_ONLY   0x328000 TO 0x32BFFF;
      PAGE_33       = READ_ONLY   0x338000 TO 0x33BFFF;
      PAGE_34       = READ_ONLY   0x348000 TO 0x34BFFF;
      PAGE_35       = READ_ONLY   0x358000 TO 0x35BFFF;
      PAGE_36       = READ_ONLY   0x368000 TO 0x36BFFF;
      PAGE_37       = READ_ONLY   0x378000 TO 0x37BFFF;
      PAGE_38       = READ_ONLY   0x388000 TO 0x38BFFF;
      PAGE_39       = READ_ONLY   0x398000 TO 0x39BFFF;
      PAGE_3A       = READ_ONLY   0x3A8000 TO 0x3ABFFF;
      PAGE_3B       = READ_ONLY   0x3B8000 TO 0x3BBFFF;
      PAGE_3C       = READ_ONLY   0x3C8000 TO 0x3CBFFF;
      PAGE_3D       = READ_ONLY   0x3D8000 TO 0x3DBFFF;
/*    PAGE_3E       = READ_ONLY   0x3E8000 TO 0x3EBFFF; not used:
equivalent to ROM_4000 */
/*    PAGE_3F       = READ_ONLY   0x3F8000 TO 0x3FBEFF; not used:
equivalent to ROM_C000 */
END

PLACEMENT /* here all predefined and user segments are placed into the
SEGMENTS defined above. */
     _PRESTART,             /* Used in HIWARE format: jump to _Startup
at the code start */
      STARTUP,                /* startup data structures */
      ROM_VAR,                /* constant variables */
      STRINGS,                /* string literals */
      VIRTUAL_TABLE_SEGMENT,  /* C++ virtual table segment */
    //.ostext,                /* OSEK */
```

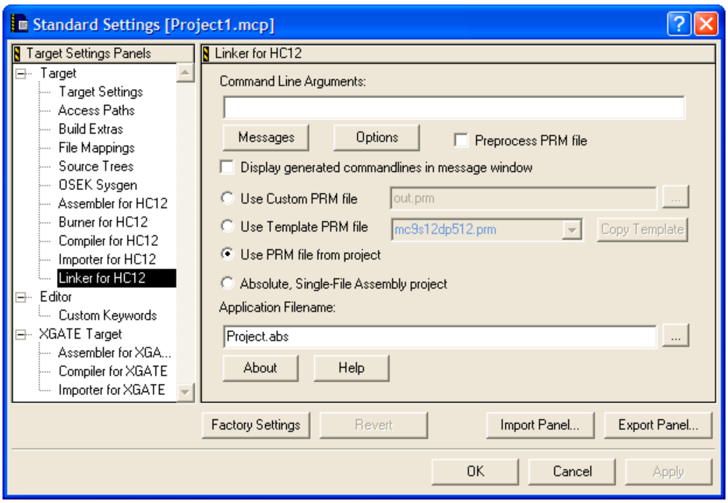
```
      NON_BANKED,             /* runtime routines which must not be
banked */
     COPY                  /* copy down information: how to initialize
variables */
                             /* in case you want to use ROM_4000 here
as well, make sure
                                 that all files (incl. library files)
are compiled with the
                                 option: -OnB=b */
                        INTO  ROM_C000/*, ROM_4000*/;

      DEFAULT_ROM       INTO  PAGE_20, PAGE_21, PAGE_22, PAGE_23,
PAGE_24, PAGE_25, PAGE_26, PAGE_27,
                              PAGE_28, PAGE_29, PAGE_2A, PAGE_2B,
PAGE_2C, PAGE_2D, PAGE_2E, PAGE_2F,
                              PAGE_30, PAGE_31, PAGE_32, PAGE_33,
PAGE_34, PAGE_35, PAGE_36, PAGE_37,
                              PAGE_38, PAGE_39, PAGE_3A, PAGE_3B,
PAGE_3C, PAGE_3D                  ;

    //.stackstart,            /* eventually used for OSEK kernel
awareness: Main-Stack Start */
      SSTACK,                 /* allocate stack first to avoid
overwriting variables on overflow */
    //.stackend,              /* eventually used for OSEK kernel
awareness: Main-Stack End */
    DEFAULT_RAM         INTO  RAM;

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

ENTRIES /* keep the following unreferenced variables */
    /* OSEK: always allocate the vector table and all dependent objects
*/
  //_vectab OsBuildNumber _OsOrtiStackStart _OsOrtiStart
END

STACKSIZE 0x100

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

initialization entry point */
```

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

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

The **Linker for HC12** preference panel allows you to select the PRM file you want to use for your CodeWarrior project. The default PRM file for a CodeWarrior project is the PRM file in the project window. To set preferences, select **Edit >** <*target_name*> **Settings > Target > Linker for HC12.** The **Linker for HC12** preference panel appears (Figure 1.31).

Figure 1.31 Linker for HC12 Preference Panel



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

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

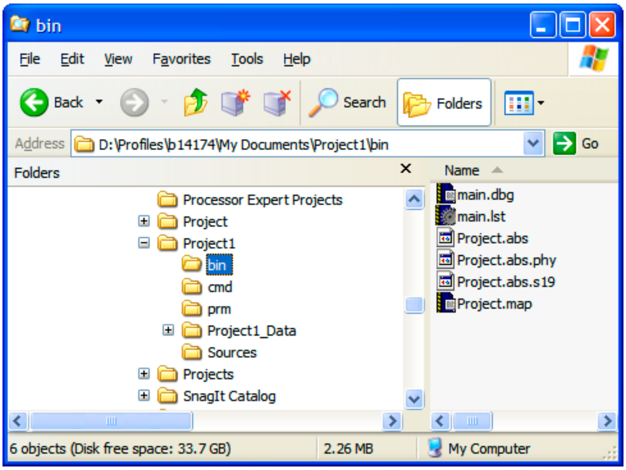
In case you want to change the filename of the application, you can determine the filename and its path with the *Application Filename:* text box. See the *Smart Linker section of the "Build Tools"* manual for details.

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

Linking Object-Code Files

You can run this relocatable assembly project from the **Project** menu: Select **Project > Make or Project > Debug.** The Linker generates a `*.abs` file and a `*.abs.s19` standard S-Record File in the `bin` subfolder of the project directory. You can use the S-Record File for programming a ROM memory (Figure 1.32).

Figure 1.32 Project Directory in Windows Explorer After Linking



NOTE The *Full Chip Simulation* option in the CodeWarrior IDE was selected when the project was created, so if **Project > Debug** is selected, the **True-Time Simulator and & Real-Time Debugger** window opens and you can follow each assembly-code instruction during the execution of the program with the Simulator.

To single-step the simulator through the program's assembly-source instructions, select **Run > Assembly Step** from the main menu or press the **Ctrl+F11** keys.

Linking with Linker

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

- Start your editor and create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as <target_name>.prm.
- Store the PRM file in a convenient location. A good spot would be directly into the project directory.
- In the <target_name>.prm file, add the name of the executable (*.abs) file, say <target_name>.abs. In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's `Fibonacci.prm` file is shown in Listing 1.6 (a PRM file for an MC9S12DP512 from another CodeWarrior project was adapted).

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

```
/* This is a linker parameter file for the MC9S12DP512 */
NAMES END /* CodeWarrior will pass all the needed files to the linker
by command line. But here you may add your own files too. */

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

/* Register space  */
/*     IO_SEG        = PAGED          0x0000 TO   0x03FF; intentionally
not defined */

/* EPROM */
      EEPROM         = READ_ONLY      0x0400 TO   0x07FF;

/* RAM */
      RAM            = READ_WRITE     0x0800 TO   0x3FFF;

/* non-paged FLASHs */
      ROM_4000       = READ_ONLY      0x4000 TO   0x7FFF;
      ROM_C000       = READ_ONLY      0xC000 TO   0xFEFF;
 /*   VECTORS        = READ_ONLY      0xFF00 TO   0xFFFF; intentionally
not defined: used for VECTOR commands below */
   //OSVECTORS       = READ_ONLY      0xFF8C TO   0xFFFF;   /* OSEK
interrupt vectors (use your vector.o) */

/* paged FLASH:                       0x8000 TO   0xBFFF; addressed
through PPAGE */
      PAGE_20        = READ_ONLY    0x208000 TO 0x20BFFF;
      PAGE_21        = READ_ONLY    0x218000 TO 0x21BFFF;
      PAGE_22        = READ_ONLY    0x228000 TO 0x22BFFF;
```

```
      PAGE_23       = READ_ONLY   0x238000 TO 0x23BFFF;
      PAGE_24       = READ_ONLY   0x248000 TO 0x24BFFF;
      PAGE_25       = READ_ONLY   0x258000 TO 0x25BFFF;
      PAGE_26       = READ_ONLY   0x268000 TO 0x26BFFF;
      PAGE_27       = READ_ONLY   0x278000 TO 0x27BFFF;
      PAGE_28       = READ_ONLY   0x288000 TO 0x28BFFF;
      PAGE_29       = READ_ONLY   0x298000 TO 0x29BFFF;
      PAGE_2A       = READ_ONLY   0x2A8000 TO 0x2ABFFF;
      PAGE_2B       = READ_ONLY   0x2B8000 TO 0x2BBFFF;
      PAGE_2C       = READ_ONLY   0x2C8000 TO 0x2CBFFF;
      PAGE_2D       = READ_ONLY   0x2D8000 TO 0x2DBFFF;
      PAGE_2E       = READ_ONLY   0x2E8000 TO 0x2EBFFF;
      PAGE_2F       = READ_ONLY   0x2F8000 TO 0x2FBFFF;
      PAGE_30       = READ_ONLY   0x308000 TO 0x30BFFF;
      PAGE_31       = READ_ONLY   0x318000 TO 0x31BFFF;
      PAGE_32       = READ_ONLY   0x328000 TO 0x32BFFF;
      PAGE_33       = READ_ONLY   0x338000 TO 0x33BFFF;
      PAGE_34       = READ_ONLY   0x348000 TO 0x34BFFF;
      PAGE_35       = READ_ONLY   0x358000 TO 0x35BFFF;
      PAGE_36       = READ_ONLY   0x368000 TO 0x36BFFF;
      PAGE_37       = READ_ONLY   0x378000 TO 0x37BFFF;
      PAGE_38       = READ_ONLY   0x388000 TO 0x38BFFF;
      PAGE_39       = READ_ONLY   0x398000 TO 0x39BFFF;
      PAGE_3A       = READ_ONLY   0x3A8000 TO 0x3ABFFF;
      PAGE_3B       = READ_ONLY   0x3B8000 TO 0x3BBFFF;
      PAGE_3C       = READ_ONLY   0x3C8000 TO 0x3CBFFF;
      PAGE_3D       = READ_ONLY   0x3D8000 TO 0x3DBFFF;
/*    PAGE_3E       = READ_ONLY   0x3E8000 TO 0x3EBFFF; not used:
equivalent to ROM_4000 */
/*    PAGE_3F       = READ_ONLY   0x3F8000 TO 0x3FBEFF; not used:
equivalent to ROM_C000 */
END

PLACEMENT /* here all predefined and user segments are placed into the
SEGMENTS defined above. */
      _PRESTART,             /* Used in HIWARE format: jump to _Startup
at the code start */
      STARTUP,                /* startup data structures */
      ROM_VAR,                /* constant variables */
      STRINGS,                /* string literals */
      VIRTUAL_TABLE_SEGMENT,  /* C++ virtual table segment */
    //.ostext,                /* OSEK */
      NON_BANKED,             /* runtime routines which must not be
banked */
      COPY                 /* copy down information: how to initialize
variables */
                             /* in case you want to use ROM_4000 here
as well, make sure
```

```
                                 that all files (incl. library files)
are compiled with the
                                 option: -OnB=b */
                        INTO  ROM_C000/*, ROM_4000*/;

      DEFAULT_ROM       INTO  PAGE_20, PAGE_21, PAGE_22, PAGE_23,
PAGE_24, PAGE_25, PAGE_26, PAGE_27,
                              PAGE_28, PAGE_29, PAGE_2A, PAGE_2B,
PAGE_2C, PAGE_2D, PAGE_2E, PAGE_2F,
                              PAGE_30, PAGE_31, PAGE_32, PAGE_33,
PAGE_34, PAGE_35, PAGE_36, PAGE_37,
                              PAGE_38, PAGE_39, PAGE_3A, PAGE_3B,
PAGE_3C, PAGE_3D                  ;

    //.stackstart,            /* eventually used for OSEK kernel
awareness: Main-Stack Start */
      SSTACK,                 /* allocate stack first to avoid
overwriting variables on overflow */
    //.stackend,              /* eventually used for OSEK kernel
awareness: Main-Stack End */
    DEFAULT_RAM         INTO  RAM;

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

ENTRIES /* keep the following unreferenced variables */
    /* OSEK: always allocate the vector table and all dependent objects
*/
  //_vectab OsBuildNumber _OsOrtiStackStart _OsOrtiStart
END

STACKSIZE 0x100

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

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

NOTE The default size for the stack using the CodeWarrior New Project Wizard for the `MC9S12DP512` is 256 bytes: (STACKSIZE 0x100). This command and `__SEG_END_SSTACK` in the assembly code file determine the size and placement of the stack in RAM:

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

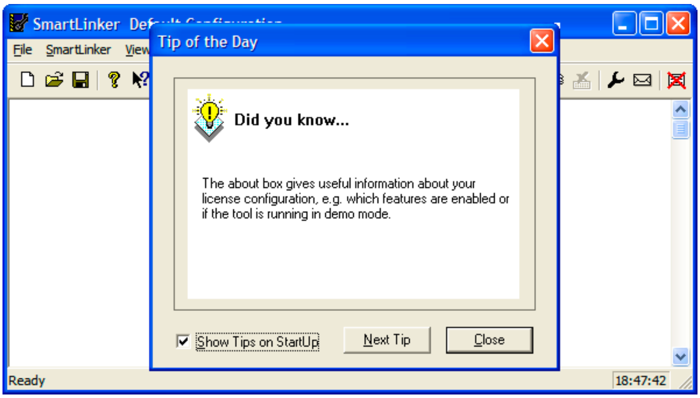
The commands in the linker parameter file are described in the Linker portion of the Build Tools manual.

To link a file, perform these steps:

1. Start the Linker. The SmartLinker tool is located in the `prog` folder in the `<CodeWarrior installation>\Prog\linker.exe.`

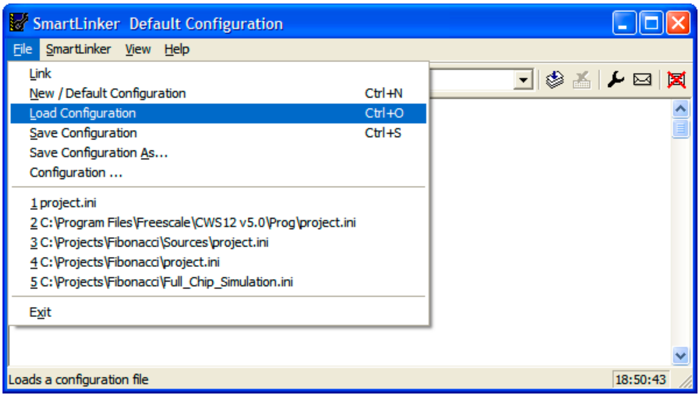
   The **SmartLinker Default Configuration** windows appears (Figure 1.36).

Figure 1.33 SmartLinker Default Configuration



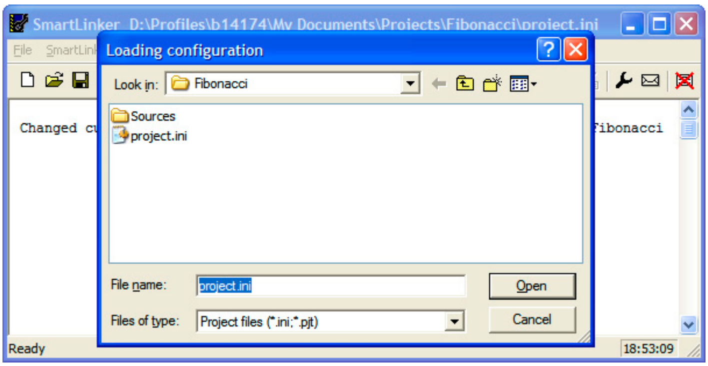
2. Click the **Close** button to close the **Tip of the Day** dialog box.
3. Select **File > Load Configuration** (Figure 1.34) to load an existing project's configuration file.

Figure 1.34 Load Configuration



4. In the **Loading configuration** dialog box, select the same `<project>.ini` that the Assembler used for its configuration - the `project.ini` file in the project directory (Figure 1.35).

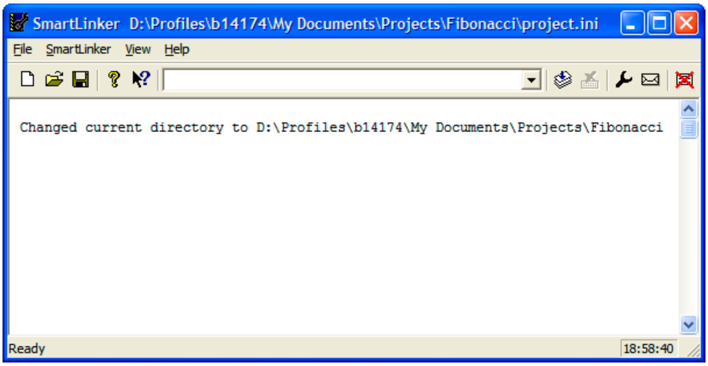
Figure 1.35 Loading configuration Dialog Box



5. Press the **Open** button to load the configuration file. The project directory is now the current directory for the Linker. You can press the **Save** button to save the configuration if you choose.

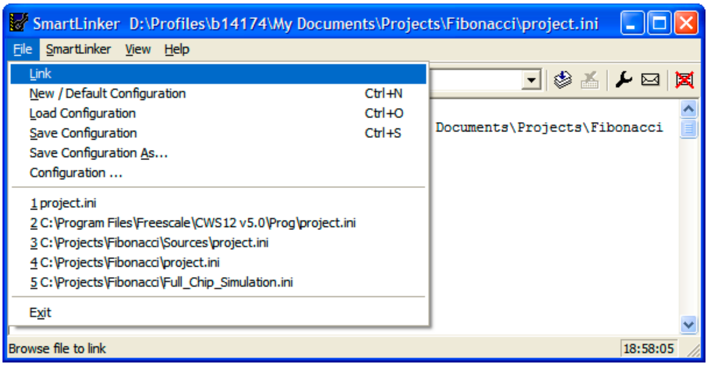
   The current directory is changed (Figure 1.36).

Figure 1.36 SmartLinker Configuration



6. Select **File > Link** (Figure 1.37).

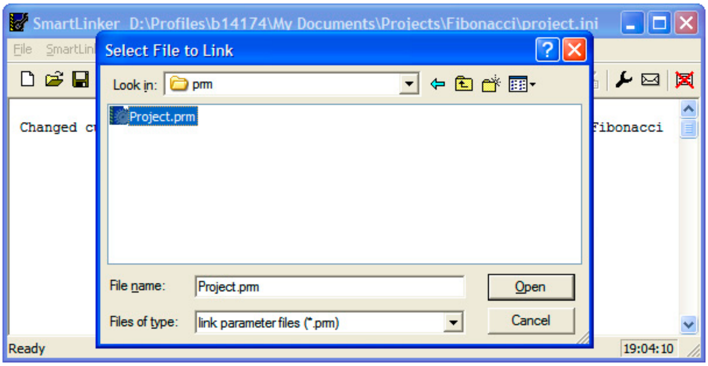
Figure 1.37 Select File to Link



The **Select Files to Link** dialog box appears (Figure 1.38).

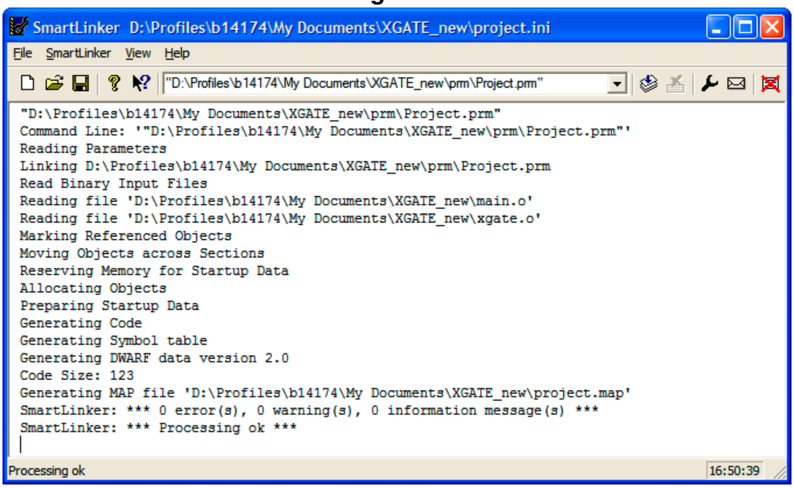
7. Browse to locate the PRM file for your project. Select the PRM file.

Figure 1.38 Select Files to Link Dialog Box



8. Press the **Open** button.
   The SmartLinker 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 1.39).

Figure 1.39 Linker Main Window After Linking



The messages in the linker's project window indicate:

- The current directory for the Linker is the project directory, `D:\Profiles\b14174\My Documents\Projects\Fibonacci`
- The `Project.prm` file was used to name the executable file, which object files were linked, and how the RAM and ROM memory areas are to be allocated for the

relocatable sections. The Reset and application entry points were also specified in this file.

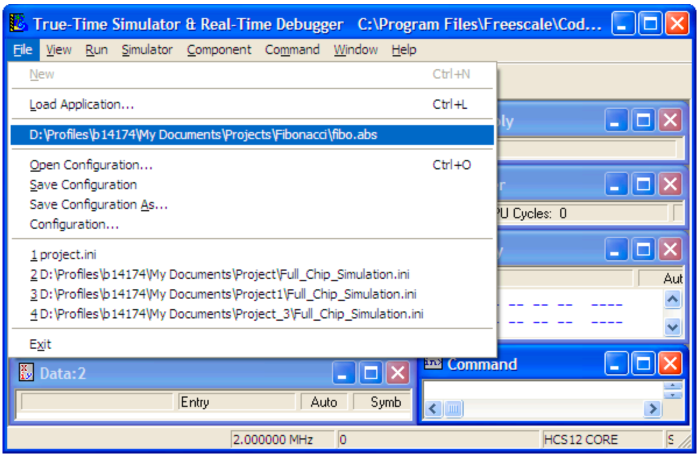
- There was one object-code file, `main.o`.
- The output format was DWARF 2.0.
- The Code Size was 46 bytes.
- A Linker Map file was generated - `Fibo.map`.
- No errors or warnings occurred and no information messages were issued.

The Simulator/Debugger Build Tool, `hiwave.exe`, located in the `prog` folder in the CodeWarrior installation could be used to simulate the Fibonacci program in the `main.asm` source-code file. To operate the Simulator Build Tool, perform these steps:

1. Start the Simulator.
2. Load the absolute executable file:
   - `File > Load Application...` and browse to the appropriate `*.abs` file, or
   - Select the given path to the executable file, if it is appropriate as in this case (Figure 1.40):

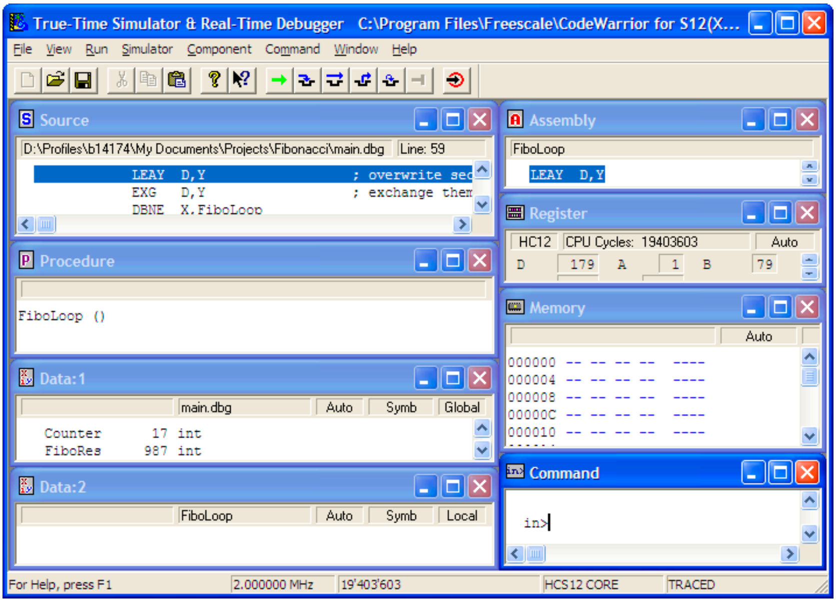
     `D:\Profiles\b14174\My Documents\Projects\Fibonacci\Fibo.abs`

Figure 1.40 Simulator: Select Executable File



3. Step (Figure 1.41) through the program source code.

Figure 1.41 Assembly Stepping



Directly Generating ABS file

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

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

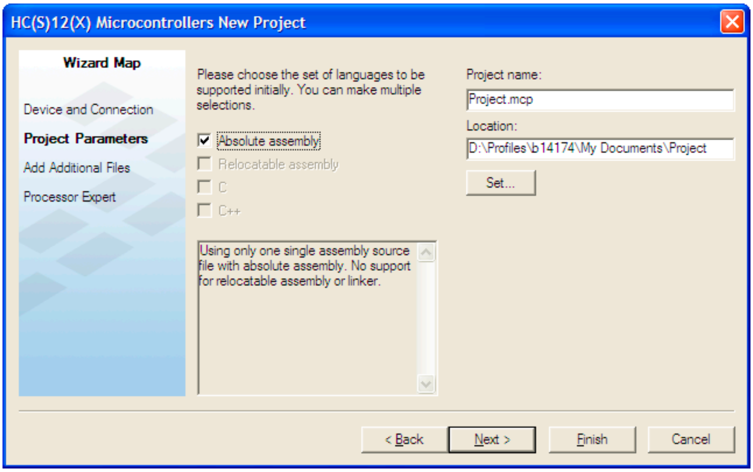
Using CodeWarrior Assembler to Generate an ABS File

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. However, there are some differences:

- No PRM file is required, so no PRM file will be included in the `Prm` group in the project window.
- Memory area allocations are determined directly in the single `*.asm` assembly source-code file.
- In the **Project Parameter** page, instead of *Relocatable Assembly* select the *Absolute Assembly* option button.

NOTE Refer the Using New Project Wizard to Create Project section, if you need assistance in creating a CodeWarrior project. However, in the Project Parameter page, select the *Absolute Assembly* option button.

Figure 1.42 Project Parameters Page - Absolute Assembly



Single Absolute-Assembly `main.asm` File

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

CAUTION We strongly recommend that you use separate sections for code, (variable) data, and constants. All sections used in the assembler application must be absolute and defined using the `ORG` directive. The addresses for constant or code sections have to be located in the ROM memory area, while the data sections have to be located in a RAM area (according to the memory map of the hardware that you intend to use).

The programmer is responsible for making sure that no section overlaps occur.

Listing 1.7 `main.asm` File - Absolute Assembly

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

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

; common defines and macros
            INCLUDE 'derivative.inc'

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

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


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

```
            BRA   mainLoop          ; restart.

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


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

Pay special attention to the following points:

- The Reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application's entry point address is created at the Reset vector address. To set the entry point of the application at address `$FFFE` on the `Entry` symbol, the following code is used (Listing 1.8):

Listing 1.8 Using ORG to set the Reset vector

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

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

Listing 1.9 Using ABSENTRY to enter the entry-point address

```
        ABSENTRY Entry
```

Assembling `main.asm` File

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

Listing 1.10 Assembler output listing file of main.asm

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

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

```
36   a00400B 070E                            BSR    CalcFibo
37   a00400D 7C08 02                         STD    FiboRes          ; s
38   a004010 FE08 00                         LDX    Counter
39   a004013 08                              INX
40   a004014 8E00 18                         CPX    #24              ; L
41   a004017 26EF                            BNE    counterLoop
42   a004019 20EA                            BRA    mainLoop         ; r
43
44                          CalcFibo:   ; Function to calculate Fi
45   a00401B CD00 00                         LDY    #$00             ; s
46   a00401E CC00 01                         LDD    #$01             ; l
47   a004021 0405 07                         DBEQ   X,FiboDone       ; l
48                          FiboLoop:
49   a004024 19EE                            LEAY   D,Y              ; o
50   a004026 B7C6                            EXG    D,Y              ; e
51   a004028 0435 F9                         DBNE   X,FiboLoop
52                          FiboDone:
53   a00402B 3D                              RTS                     ; r
54
55
56                          ;*************************************
57                          ;*                 I
58                          ;*************************************
59                                           ORG    $FFFE
60   a00FFFE 4000                            DC.W   Entry            ; R
```

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

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

Using Assembler for Absolute Assembly

Create a new configuration *project.ini* file and directory for the absolute assembly project using the standalone Assembler Build Tool. Use an absolute assembly source file of the type listed in Listing 1.11.

Listing 1.11 Main.asm file for absolute assembly

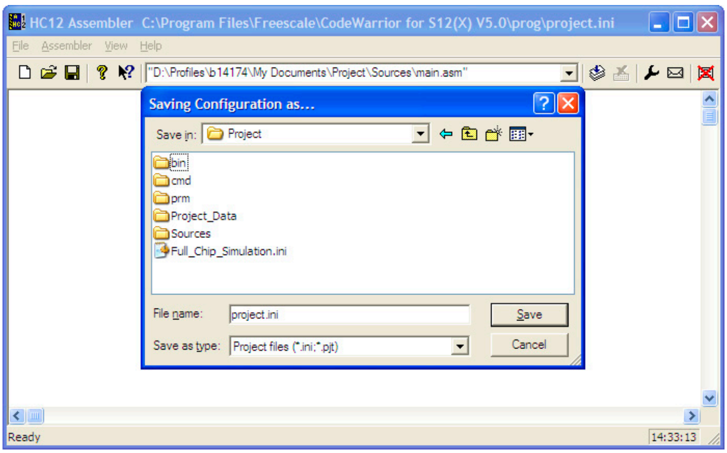
```
;*****************************************************************
;* This stationery serves as the framework for a                 *
;* user application (single file, absolute assembly application) *
;*****************************************************************

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

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

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

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


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

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

```
            DBNE  X,FiboLoop
FiboDone:
            RTS                   ; result in D

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

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

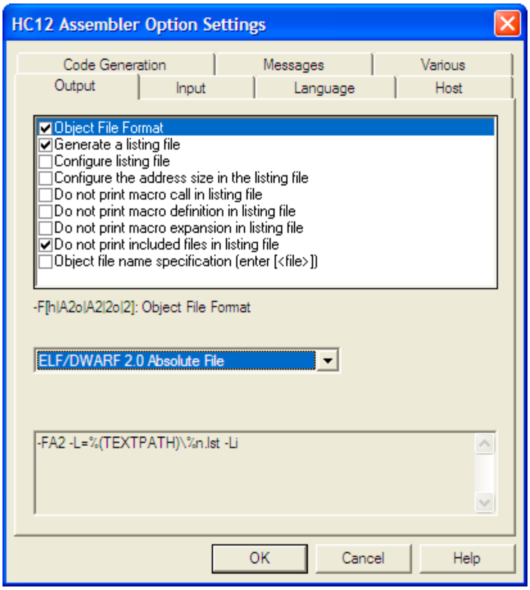
1. Start the Assembler. You can do this by opening the `ahc12.exe` file in the `prog` folder in the S12(X) CodeWarrior installation. The Assembler opens. Close the *Tip of the Day* dialog box if this dialog box is open.
2. Create a new `project.ini` configuration file (**File > New / Default Configuration**) and store it in the project directory (**File > Save Configuration As**) This makes the project directory the current directory for the Assembler (Figure 1.43).

Figure 1.43 Creating a New Absolute Assembly Project



3. Select **Assembler > Options**. The **HC12 Assembler Option Settings** dialog box appears (Figure 1.44).

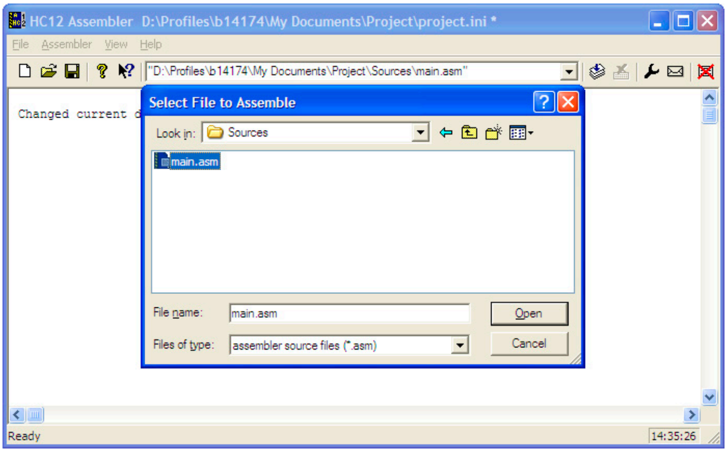
Figure 1.44 HC12 Assembler Option Settings Dialog Box



4. In the **Output** panel, check the *Object File Format* checkbox. The Assembler displays more information at the bottom of the dialog box. Select the *ELF/DWARF 2.0 Absolute File* option button. The assembler options for generating a listing file can also be set at this point, if desired. Click the **OK** button.

5. Select the assembly source-code file that will be assembled: Select **File > Assemble.** The **Select File to Assemble** dialog box appears (Figure 1.45).

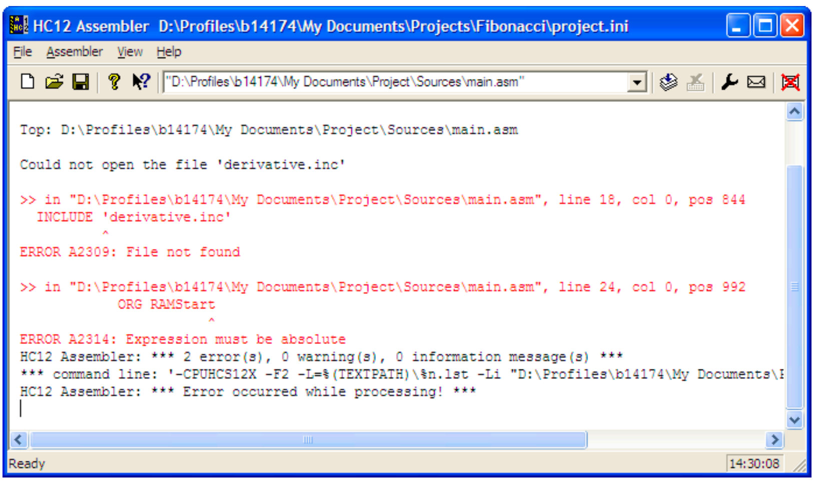
Figure 1.45 Select File to Assemble Dialog Box



6. Browse to the assembly source-code file. Click the **Open** button.

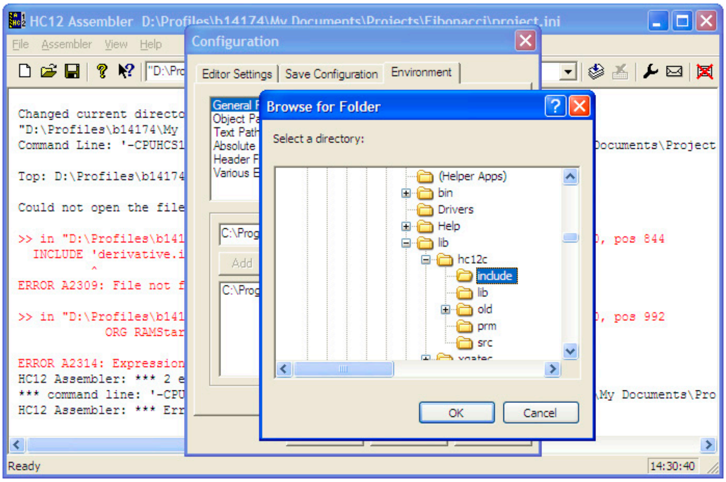
   The Assembler now assembles the source code. Error-message (Figure 1.46) or positive feedback about the assembly process is created in the assembler main window.

Figure 1.46 "ERROR A2309: File not found" Error Message



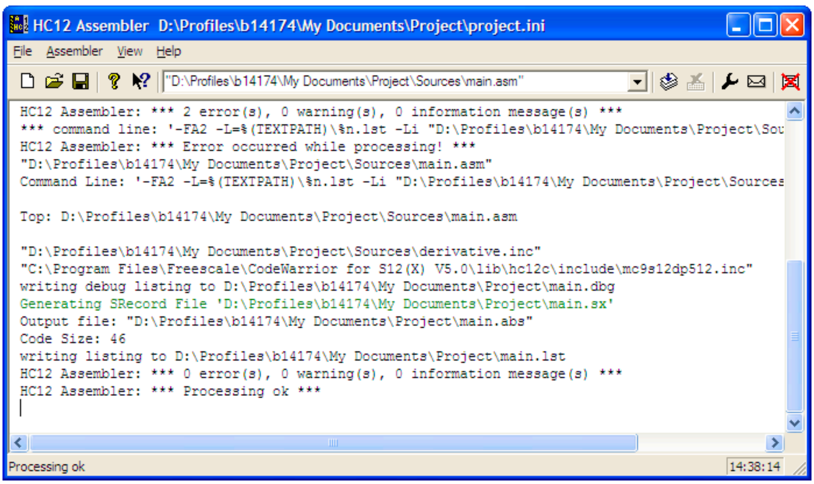
7. Make sure that the `GENPATH` configuration is set (Figure 1.47) for the include file used by the `main.asm` file in this project in the event an error message for a missing file appears, as above.

Figure 1.47 Adding a GENPATH for the include file



8. Select **File > Configuration... > Environment > General Path** and browse for the missing include file. (See Adding GENPATH for instructions for setting a GENPATH.) After setting a GENPATH to the folder of the include file, try assembling again.
9. Select **File > Assemble** and browse for the `*.asm` file and click the **Open** button for the assembly command. This time, it should assemble correctly (Figure 1.48).

Figure 1.48 Successful Absolute Assembly



The messages indicate that:

- An assembly source code (`main.asm`) file and an `MC9S12DP512.inc` file were read as input.
- A debugging `(main.dbg)` file was generated in the project directory.
- An S-Record File (`main.sx)` was created. This file can be used to program ROM memory.
- An absolute executable file was generated, `main.abs`.
- The Code Size was 46 bytes.

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

2

Assembler Graphical User Interface

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

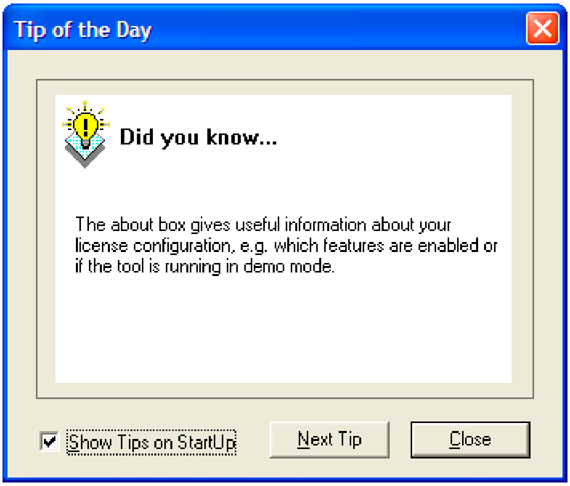
This chapter covers the following topics:

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

Starting Assembler

When you start the Assembler, the Assembler displays a standard *Tip of the Day* (Figure 2.1) dialog box containing news and tips about the Assembler.

Figure 2.1 Tip of the Day Dialog Box



1. Click the **Next Tip** button to see the next piece of information about the Assembler.
2. Click the **Close** button to close the **Tip of the Day** dialog box.

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

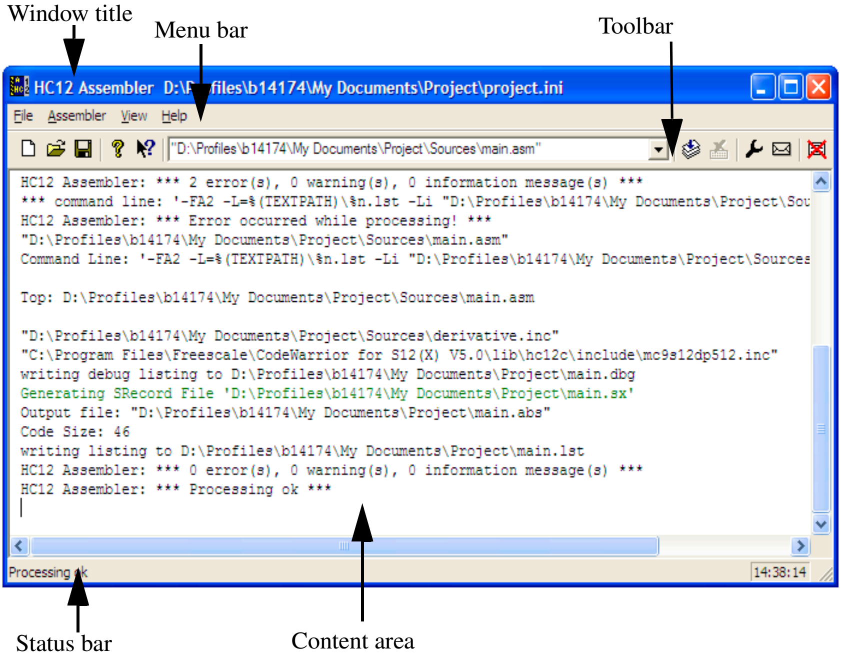
NOTE If you want the Assembler to automatically open the standard **Tip of the Day** dialog box at Assembler start up, choose **Help > Tip of the Day**. The Assembler displays the **Tip of the Day** dialog box. Check the Show Tips on StartUp checkbox.

Assembler Main Window

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

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

Figure 2.2 Assembler Main Window



Window Title

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

Content Area

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

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

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

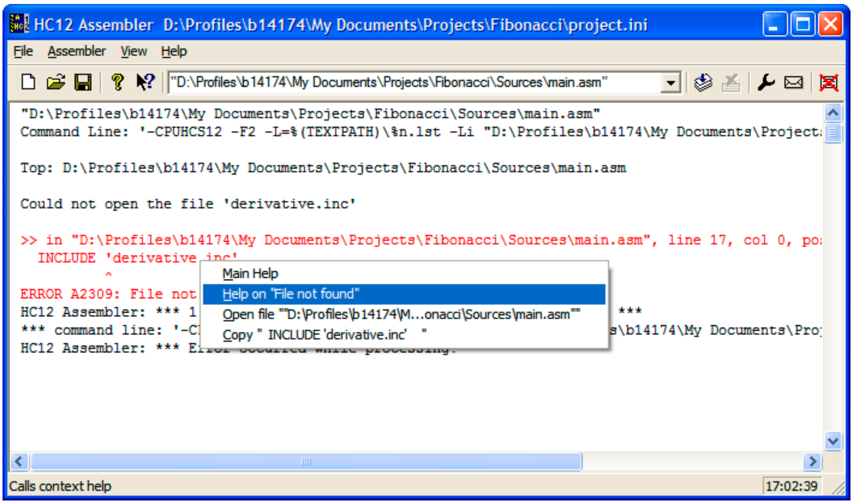
All text in the assembler window content area can have context information consisting of two items:

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

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

- If a file context is available for a line, double-click on a line containing file-context information.
- Click with the right mouse on the line and select *Open...*. This entry is only available if a file context is available (Figure 2.3).

Figure 2.3 Right-Context Help



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

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

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

Toolbar

Figure 2.4 displays the elements of the Toolbar.

Figure 2.4 Toolbar

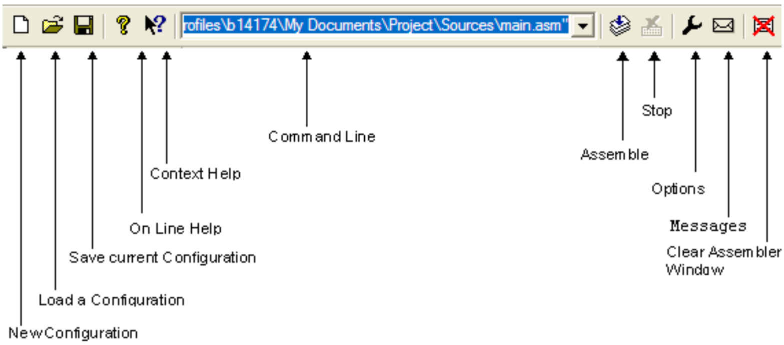
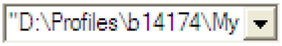


Table 2.1 describes the Assembler toolbar elements.

Table 2.1 Assemble Toolbar Elements

Element	Decription
	New — Click to load the default configuration.
	Load — Click to select and load a configuration file.
	Save — Click to save the current configuration.
	Help — Click to invoke the help file.

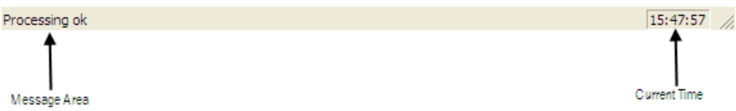
Table 2.1 Assemble Toolbar Elements (*continued*)

Element	Decription
	Context Help — Click to invoke the context help.
	Editable combo box — Click to display the list of commands which were executed.
	Assemble — Click to execute command line.
	Stop — Click to stop the current assembly session; is enabled when some file is assembled.
	Options — Click to open the **Option Settings** dialog box.
	Message — Click to open the **Message Settings** dialog box.
	Clear — Click to clear the assembler window's content area.

Status Bar

Figure 2.5 displays the elements of the Status bar.

Figure 2.5 Status bar



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

Assembler Menu Bar

The following menus are available in the menu bar (Table 2.2):

Table 2.2 Menu bar options

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

File Menu

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

- the assembler option settings specified in the assembler dialog boxes,
- the list of the last command line which was executed and the current command line,
- the window position, size, and font,
- the editor currently associated with the Assembler. This editor may be specifically associated with the Assembler or globally defined for all *Tools*. (See Editor Settings Dialog Box.),
- 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 2.3 File Menu Options

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

Assembler Menu

The Assembler menu (Table 2.4) allows you to customize the Assembler. You can graphically set or reset the Assembler options or to stop the assembling process.

Table 2.4 Assembler Menu Options

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

View Menu

The View menu (Table 2.5) 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 2.5 View Menu Options

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

Editor Settings Dialog Box

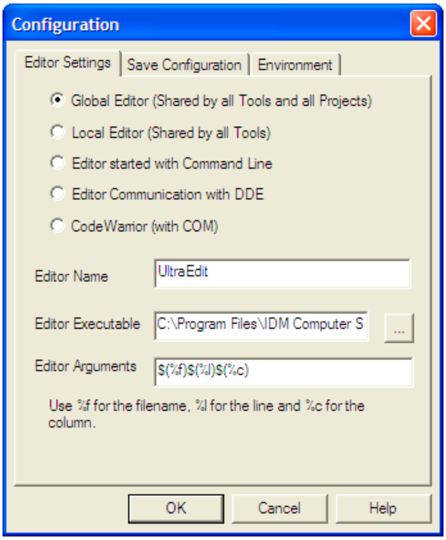
The **Editor Setting** dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

These are the following main entries:

Global Editor (Shared by all Tools and Projects)

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

Figure 2.6 Global Editor Configuration

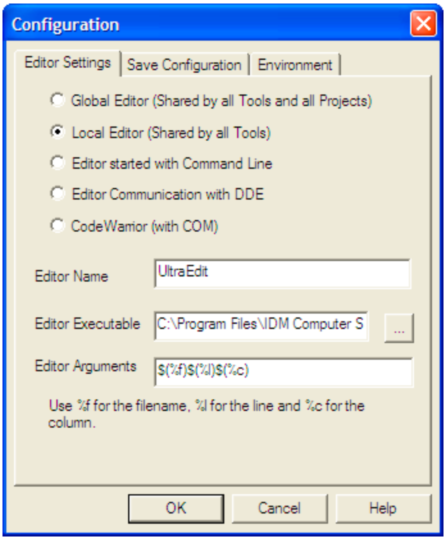


Local Editor (Shared by all Tools)

This entry (Figure 2.7) is shared by all tools for the current project. This setting is stored in the `[Editor]` section of the local initialization file, usually `project.ini` in the current directory. Some Modifiers can be specified in the editor command line.

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

Figure 2.7 Local Editor Configuration



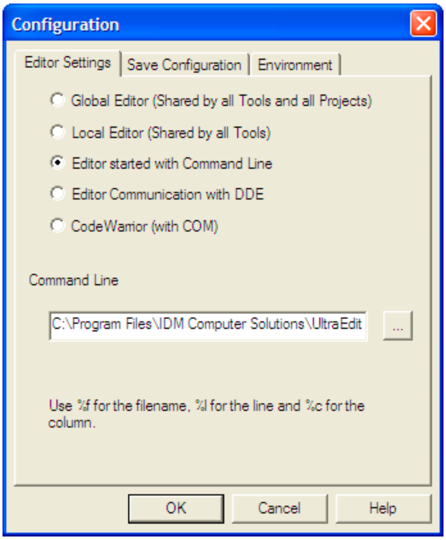
Editor Started with the Command Line

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

Enter the command which should be used to start the editor.

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

Figure 2.8 Command-Line Editor Configuration



Examples of Configuring a Command-Line Editor

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

Listing 2.1 CodeWright editor configuration

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

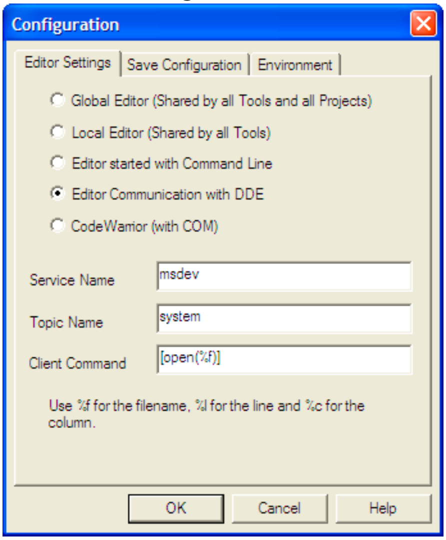
Listing 2.2 WinEdit editor configuration

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

Editor Started with DDE

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

Figure 2.9 DDE Editor Configuration



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

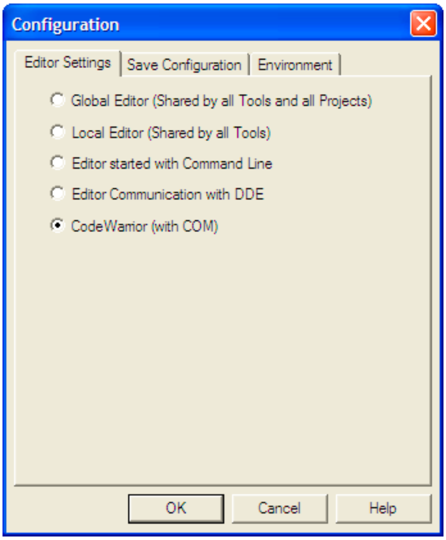
Listing 2.3 Microsoft Developer Studio Configuration Settings

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

CodeWarrior with COM

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

Figure 2.10 COM Editor Configuration



Modifiers

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

- The `%f` modifier refers to the name of the file (including path and extension) where the error has been detected.
- The `%l` modifier refers to the line number where the message has been detected.
- The `%c` modifier refers to the column number where the message has been detected.

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

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

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

Save Configuration Dialog Box

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

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

This dialog box has the following configurations:

- *Options*: This item is related to the option and message settings. If this checkbox is set, the current option and message settings are stored in the project file when the configuration is saved. By disabling this checkbox, 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 checkbox, the current editor settings are stored in the project file when the configuration is saved. If you disable this checkbox, 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 checkbox, these settings are stored in the project file when the current configuration is saved. If you disable this checkbox, 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 is used to configure the environment. The content of the dialog is read from the actual project file out of the [Environment Variables] section.

The following variables are available:

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

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

The following buttons are available:

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

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

Option Settings Dialog Box

This dialog box allows you to set/reset assembler options. The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet (Table 2.6):

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

An assembler option is set when the checkbox 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 dialog box to map messages to a different message class.

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

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.
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):

Table 2.8 Types of Message Groups

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

Table 2.8 Types of Message Groups (*continued*)

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

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

Changing the Class Associated with a Message

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

Example:

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

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

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

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

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

About Dialog Box

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

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

Click on *OK* to close this dialog box.

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

Specifying the Input File

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

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

Use the Command Line in the Toolbar to Assemble

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

Assembling a New File

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

Assembling an Existing Assembled File

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

Use the File > Assemble... Entry

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

Use Drag and Drop

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

Message/Error Feedback

After assembly, there are several ways to check where different errors or warnings have been detected. The default format of the error message is as Listing 2.4.

Listing 2.4 Default configuration of an error message

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

A typical error message is like the one in Listing 2.5.

Listing 2.5 Error message example

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

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

For different message formats, see the following Assembler options:

- -WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode,
- -WmsgFob: Message format for batch mode,
- -WmsgFoi: Message format for interactive mode,
- -WmsgFonf: Message format for no file information, and
- -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 an User-Defined Editor

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

Line Number can be Specified on the Command Line

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

Line Number cannot be Specified on the Command Line

Editors like *WinEdit v31* or lower, *Notepad*, or *Wordpad* cannot be started with a line number in the command line. When these editors have been correctly configured, they can be started automatically by double clicking on an error message. The configured editor will be started, and the file is automatically opened where the error occurs. To scroll to the position where the error was detected, you have to:

- Activate the assembler again.
- Click the line on which the message was generated. This line is highlighted on the screen.

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

3

Environment

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

There are three ways to specify an environment:

1. The current project file with the Environment Variables section. This file may be specified on Tool startup using the -Prod: Specify project file at startup assembler option. This is the recommended method and is also supported by the IDE.
2. An optional `default.env` file in the current directory. This file is supported for compatibility reasons with earlier versions. The name of this file may be specified using the ENVIRONMENT: Environment file specification environment variable. Using the `default.env` file is not recommended.
3. Setting environment variables on system level (DOS level). This is also not recommended.

Various parameters of the Assembler may be set in an environment using environment variables. The syntax is always the same (Listing 3.1).

Listing 3.1 Syntax for setting environment variables

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

Listing 3.2 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., Make Utility).

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

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

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

To overwrite this behavior, the DEFAULTDIR: Default current directory 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) 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, `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 are allowed. Note that they are case-sensitive and always surrounded by `{}`. Also the variable content contains a directory separator '\' as well.

- {Compiler}

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

- {Project}

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

- {System}

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

Global Initialization File - `mcutools.ini` (PC only)

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

Listing 3.4 shows two typical locations used for the `mcutools.ini` files.

Listing 3.4 Usual locations for the mcutools.ini files

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

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

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

Local Configuration File (Usually `project.ini`)

The Assembler does not change the `default.env` file in any way. The Assembler only reads the contents. All the configuration properties are stored in the configuration file. The same configuration file can and is intended to be used by different applications (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 environment variable are reloaded and added to the project's options.

This behavior has to be noticed when in different directories different `default.env` files exist which contain incompatible options in their `ASMOPTIONS` environment variables. When a project is loaded using the first `default.env` file, its `ASMOPTIONS` options are added to the configuration file. If this configuration is then stored in a different directory, where a `default.env` file exists with these incompatible options, the

Assembler adds the options and remarks the inconsistency. Then a message box appears to inform the user that those options from the default.env file were not added. In such a situation, the user can either remove the options from the configuration file with the advanced option dialog box or he can remove the option from the default.env file with the shell or a text editor depending upon which options should be used in the future.

At startup, the configuration stored in the project.ini file located in the current directory is loaded.

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

Paths

Most environment variables contain path lists telling where to look for files. A path list is a list of directory names separated by semicolons following the syntax in Listing 3.5.

Listing 3.5 Syntax used for setting path lists of environment variables

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

Listing 3.6 is a typical example of setting an environment variable.

Listing 3.6 Setting the paths for the GENPATH environment variable

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

If a directory name is preceded by an asterisk (*), the programs recursively search that whole directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list. Listing 3.7 shows the use of an asterisk (*) for recursively searching the entire C drive for a configuration file with a \INSTALL\LIB path.

Listing 3.7 Recursive search for a configuration file

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

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

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

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

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

Line Continuation

It is possible to specify an environment variable in an environment file (`default.env` or `.hidefaults`) over multiple lines using the line continuation character '\' (Listing 3.8):

Listing 3.8 Using multiple lines for an environment variable

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

Listing 3.8 is the same as the alternate source code in Listing 3.9.

Listing 3.9 Alternate form of Listing 3.8

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

But this feature may be dangerous when used together with paths. You can include a path using the line continuation character:

```
GENPATH=.\
```

```
TEXTFILE=.\txt
```

This gives this result:

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

To avoid such problems, we recommend that you use a semicolon `';'` at the end of a path if there is a backslash `'\'` at the end (Listing 3.10).

Listing 3.10 Recommended style whenever a backslash is present

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

Environment Variable Details

The remainder of this section is devoted to describing each of the environment variables available for the Assembler. The environment variables are listed in alphabetical order and each is divided into several sections (Table 3.1).

Table 3.1 Topics Used for Describing Environment Variables

Topic	Description
Tools	Lists tools which use this variable.
Synonym (where one exists)	Synonyms exist for some environment variables. These synonyms may be used for older releases of the Assembler but are deprecated and 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 `default.env` for the PC or in `.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]

COPYRIGHT: Copyright entry in object file

Tools

Compiler, Assembler, Linker, or Librarian

Syntax

`COPYRIGHT=<copyright>`

Arguments

`<copyright>`: copyright entry

Description

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

Example

`COPYRIGHT=Copyright`

See also

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

DEFAULTDIR: Default current directory

Tools

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

Syntax

`DEFAULTDIR=<directory>`

Arguments

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

Description

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

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

Example

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

See also

Current Directory

Global Initialization File - mcutools.ini (PC only)

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

Listing 3.11 Naming an error file

```
ERRORFILE=MyErrors.err
```

Listing 3.12 lists all errors into the `errors` file in the `\tmp` directory.

Listing 3.12 Naming an error file in a specific directory

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

Listing 3.13 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 3.13 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 3.14).

Listing 3.14 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 3.15).

Listing 3.15 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 3.16) shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called `EDOUT`:

Listing 3.16 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 sure to set this variable if using the WinEdit Editor, to ensure that the editor can 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

- USERNAME: User Name in object file

OBJPATH: Object file path

Tools

Compiler, Assembler, Linker, or Decoder

Syntax

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

Arguments

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

Description

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

Example

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

SRECORD: S-Record type

Tools

Assembler, Linker, or Burner

Syntax

```
SRECORD=<RecordType>
```

Arguments

`<RecordType>`: Forces the type for the S-Record File which must be generated. This parameter may take the value `'S1'`, `'S2'`, or `'S3'`.

Description

This environment variable is only relevant when absolute files are directly generated by the Macro Assembler instead of object files. When this environment variable is defined, the Assembler will generate an S-Record File containing records from the specified type (`S1` records when `S1` is specified, `S2` records when `S2` is specified, and `S3` records when `S3` is specified).

NOTE If the `SRECORD` environment variable is set, it is the user's responsibility to specify the appropriate S-Record File type. If you specify `S1` while your code is loaded above `0xFFFF`, the S-Record File generated will not be correct because the addresses will all be truncated to 2-byte values.

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

Example

```
SRECORD=S2
```

TEXTPATH: Text file path

Tools

Compiler, Assembler, Linker, or Decoder

Syntax

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

Arguments

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

Description

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

Example

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

TMP: Temporary directory

Tools

Compiler, Assembler, Linker, Debugger, or Librarian

Syntax

`TMP=<directory>`

Arguments

`<directory>`: Directory to be used for temporary files

Description

If a temporary file has to be created, normally the ANSI function `tmpnam()` is used. This library function stores the temporary files created in the directory specified by this environment variable. If the variable is empty or does not exist, the current directory is used. Check this variable if you get an error message "*Cannot create temporary file*".

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

Example

`TMP=C:\TEMP`

See also

Current Directory

USERNAME: User Name in object file

Tools

Compiler, Assembler, Linker, or Librarian

Syntax

`USERNAME=<user>`

Arguments

`<user>`: Name of user

Description

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

Example

```
USERNAME=PowerUser
```

See also

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

4

Files

This chapter covers:

- Input Files
- Output Files
- File Processing

Input Files

Input files to the Assembler:

- Source Files
- Include Files

Source Files

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

Include Files

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

Output Files

Output files from the Assembler:

- Object Files
- Absolute Files
- S-Record Files
- Listing Files
- Debug Listing Files
- Error Listing File

Object Files

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

Absolute Files

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

S-Record Files

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

- If `SRECORD` = `S1`, the S-Record File gets the `*.s1` extension.
- If `SRECORD` = `S2`, the S-Record File gets the `*.s2` extension.
- If `SRECORD` = `S3`, the S-Record File gets the `*.s3` extension.

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

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

Listing Files

After successful assembling session, the Macro Assembler generates a listing file containing each assembly instruction with their associated hexadecimal code. This file is always generated when the -L: Generate a listing file 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), 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 Closed)

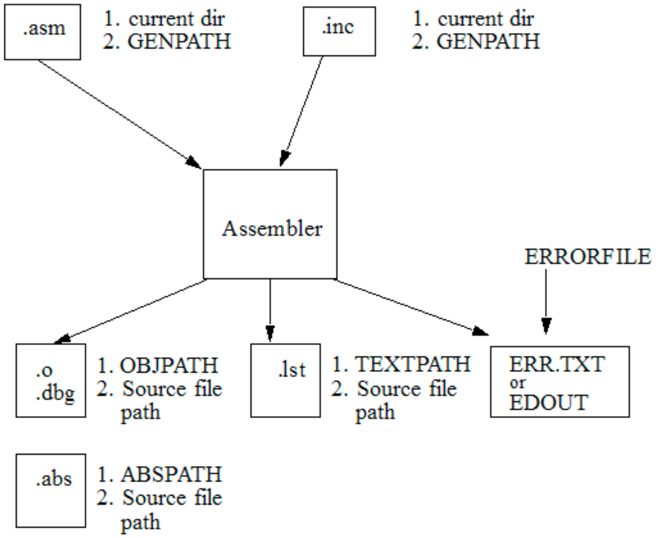
If `ERRORFILE` is set, the Assembler creates a message file named as specified in this environment variable.

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

File Processing

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

Figure 4.1 File Processing with the Assembler



5

Assembler Options

Types of Assembler Options

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

Command-line options are not case-sensitive. For example, `-L` is the same as `-li`. It is possible to coalesce 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 also creates 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) 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 (e.g., ANSI-C, C++)
Host	Lists options related to the host.
Code Generation	Lists options related to code generation (e.g., 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).

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).

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).

Table 5.5 Special Modifiers for Assembler Options

Modifier	Description
%p	Path including file separator
%N	Filename in strict 8.3 format
%n	Filename without its extension
%E	Extension in strict 8.3 format

Table 5.5 Special Modifiers for Assembler Options (*continued*)

Modifier	Description
%e	Extension
%f	Path + filename without its extension
%”	A double quote (“) if the filename, the path or the extension contains a space
%’	A single quote (‘) if the filename, the path, or the extension contains a space
%(ENV)	Replaces it with the contents of an environment variable
%%	Generates a single ‘%’

Examples Using Special Modifiers

The assumed path and filename (filename base for the modifiers) used for the following examples is displayed in Listing 5.1.

Listing 5.1 Example filename and path used for the following examples

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

Using the `%p` modifier shows the path with a file separator but without the filename.

`C:\Freescale\my demo\`

Using the `%N` modifier only displays the filename in 8.3 format but without the file extension.

`TheWhole`

The `%n` modifier returns the entire filename but with no file extension.

`TheWholeThing`

Using `%E` as a modifier returns the first three characters in the file extension.

`myE`

If you want the entire file extension, use the `%e` modifier.

`myExt`

The `%f` modifier returns the path and the filename but without the file extension.

`C:\Freescale\my demo\TheWholeThing`

The path in Listing 5.1 contains a space, therefore using `%"` or `%'` is recommended.

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

or

`'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 5.2, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.3.

Listing 5.2 Example for setting TEXTPATH

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

Listing 5.3 $(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:

`myfile.txt`

It is also possible to display the percent sign by using %%. %e%% allows the expression of a percent sign after the extension:

`myExt%`

List of Assembler Options

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

Table 5.6 Assembler Options

Assembler Options
-Ci: Switch case sensitivity on label names OFF
-CMacAngBrack: Angle brackets for grouping Macro Arguments
-CMacBrackets: Square brackets for macro arguments grouping
-Compat: Compatibility modes

Table 5.6 Assembler Options (*continued*)

Assembler Options
-CpDirect: Define DIRECT register value
-Cpu (-CpuCPU12, -CpuHCS12, -CpuHCS12X): Derivative
-D: Define Label
-Env: Set environment variable
-F (-Fh, -F2o, -FA2o, -F2, -FA2): Output file format
-H: Short Help
-I: Include file path
-L: Generate a listing file
-Lasmc: Configure listing file
-Lasms: Configure the address size in the listing file
-Lc: No macro call in listing file
-Ld: No macro definition in listing file
-Le: No macro expansion in listing file
-Li: Not included file in listing file
-Lic: License information
-LicA: License information about every feature in directory
-LicBorrow: Borrow license feature
-LicWait: Wait until floating license is available from floating License Server
-MacroNest: Configure maximum macro nesting
-M (-Ms, -Mb, -Ml): Memory Model
-MCUasm: Switch compatibility with MCUasm ON
-N: Display notify box
-NoBeep: No beep in case of an error
-NoDebugInfo: No debug information for ELF/DWARF files
-NoEnv: Do not use environment

Table 5.6 Assembler Options (*continued*)

Assembler Options
-ObjN: Object filename specification
-Prod: Specify project file at startup
-Struct: Support for structured types
-V: Prints the Assembler version
-View: Application standard occurrence
-W1: No information messages
-W2: No information and warning messages
-WErrFile: Create "err.log" error file
-Wmsg8x3: Cut filenames in Microsoft format to 8.3
-WmsgCE: RGB color for error messages
-WmsgCF: RGB color for fatal messages
-WmsgCI: RGB color for information messages
-WmsgCU: RGB color for user messages
-WmsgCW: RGB color for warning messages
-WmsgFb (-WmsgFbv, -WmsgFbm): Set message file format for batch mode
-WmsgFi (-WmsgFiv, -WmsgFim): Set message file format for interactive mode
-WmsgFob: Message format for batch mode
-WmsgFoi: Message format for interactive mode
-WmsgFonf: Message format for no file information
-WmsgFonp: Message format for no position information
-WmsgNe: Number of error messages
-WmsgNi: Number of Information messages
-WmsgNu: Disable user messages
-WmsgNw: Number of warning messages
-WmsgSd: Setting a message to disable

Table 5.6 Assembler Options (*continued*)

Assembler Options
-WmsgSe: Setting a message to error
-WmsgSi: Setting a message to information
-WmsgSw: Setting a message to warning
-WOutFile: Create error listing file
-WStdout: Write to standard output

Detailed Listing of all Assembler Options

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

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

Description

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

Syntax

```
-C=SAvocet
```

Group

Various

Scope

Assembly Unit

Arguments

None

Default

none.

Example

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

See also

Semi-Avocet Compatibility

-Ci: Switch case sensitivity on label names OFF

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.

Syntax

```
-Ci
```

Group

Input

Scope

Assembly Unit

Arguments

None

Default

None

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 5.4.

Listing 5.4 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

-CMacAngBrack: Angle brackets for grouping Macro Arguments

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.

Syntax

`-CMacAngBrack(ON|OFF)`

Group

Language

Scope

Application

Arguments

`ON` or `OFF`

Default

None

See also

Macro Argument Grouping

-CMacBrackets: Square brackets for macro arguments grouping

-CMacBrackets: Square brackets for macro arguments grouping

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.

Syntax

`-CMacBrackets(ON|OFF)`

Group

Language

Scope

Application

Arguments

`ON` or `OFF`

Default

`ON`

See also

Macro Argument Grouping

-CMacAngBrack: Angle brackets for grouping Macro Arguments

-Compat: Compatibility modes

Description

This option controls some compatibility enhancements of the Assembler. The goal is not to provide 100% compatibility with any other Assembler but to make it possible to reuse as much as possible. The various suboptions control different parts of the assembly:

- =: Operator `!=` means equal

The Assembler takes the default value of the `!=` operator as *not equal*, as it is in the C language. For compatibility, this behavior can be changed to *equal* with this option. Because the danger of this option for existing code, a message is issued for every `!=` which is treated as *equal*.

- `!`: Support additional `!` operators

  The following additional operators are defined when this option is used:

  – `!^`: exponentiation

  – `!m`: modulo

  – `!@`: signed greater or equal

  – `!g`: signed greater

  – `!%`: signed less or equal

  – `!t`: signed less than

  – `!$`: unsigned greater or equal

  – `!S`: unsigned greater

  – `!&`: unsigned less or equal

  – `!l`: unsigned less

  – `!n`: one complement

  – `!w`: low operator

  – `!h`: high operator

NOTE The default values for the following `!` operators are defined:
`!.`: binary AND
`!x`: exclusive OR
`!+`: binary OR

- `c`: Alternate comment rules

  With this suboption, comments implicitly start when a space is present after the argument list. A special character is not necessary. Be careful with spaces when this option is given because part of the intended arguments may be taken as a comment. However, to avoid accidental comments, the Assembler issues a warning if such a comment does not start with a `"*"` or a `";"`.

- `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 `XDEF`s and `XREF`s. They correspond to `XREF.B` or `XDEF.B` with the same symbols without the prefix.

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

With this suboption, an otherwise improper character recognized from the line-feed character is ignored.

- `$`: Support the $ character in symbols

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

- `a`: Add some additional directives

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

  The Assembler actually supports a `SECT` directive as an alias of the usual SECTION - Declare relocatable section 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.

Syntax

`-Compat[={!|=|c|s|f|$|a|b}`

Group

Language

Scope

Application

Arguments

See description.

Default

None

Examples

Listing 5.5 demonstrates that when `-Compat=c`, comments can start with an asterisk, `*`.

Listing 5.5 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 5.6 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 5.6 Implicit comment start after a space

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

-CpDirect: Define DIRECT register value

Description

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

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

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

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

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

Syntax

`-CpDirect<num>`

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

Group

Code Generation

Scope

Application

Arguments

`<num>`

Default

`-CpDirect0x0000`

Example

Consider the following code in Listing 5.7:

Listing 5.7 Example assembly code

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

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

By default, or with `-CpDirect0x0000` option, the following assembler listing is generated (Listing 5.8):

Listing 5.8 Default assembler output listing or when using the -CpDirect0x0000 option

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

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

When using the -CpDirect0x1100 option (with the DIRECT page register contains 0x11), the assembler output listing (Listing 5.9) is generated.

Listing 5.9 Assembler output listing when using the -CpDirect0x1100 option

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

                     MyCode:    SECTION
                     Entry:
  000000 CF0A FE                LDS   #$AFE ; init Stack Pointer
  000003 8601                   LDAA  #$01
  000005 7A00 50     main:      STAA  data
  000008 5A50                   STAA  $1150
  00000A 20F9                   BRA   main
```

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

Description

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

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

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

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

Syntax

`-Cpu{CPU12|HCS12|HCS12X}`

Group

Code Generation

Scope

Application

Arguments

None

Default

`-CpuCPU12`

Examples

Consider the source code in Listing 5.10:

Listing 5.10 Example assembly code

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

Using the default or with -CpuCPU12 assembler option, the Assembler generates the output listing in Listing 5.11:

Listing 5.11 Assembler output listing when using the default or the -CpuCPU12 option

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

With the `-CpuHCS12` or the `-CpuHCS12X` option, the Assembler generates the output listing in Listing 5.12:

Listing 5.12 Assembler output listing when using the -CpuHCS12 or the -CpuHCS12X option

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

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

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

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

Consider the source code in Listing 5.13:

Listing 5.13 Example assembly code

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

When using the `-CpuHCS12X` option, the Assembler generates the output listing in Listing 5.14:

Listing 5.14 Assembler output listing when using the -CpuHCS12X option

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

See also

CPU12 Reference Manual, paragraph 3.9.1 Move Instructions

-D: Define Label

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.

Syntax

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

Group

Input

Scope

Assembly Unit

Arguments

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

Default

`0` for `Value`.

Example

Conditional inclusion of a copyright notice. See Listing 5.15 and Listing 5.16.

Listing 5.15 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 " 2009-"
   YearAsString YEAR
 endif
   DC.B 0
 endif
```

When assembled with the `-dADD_COPYRIGHT -dYEAR=2009` option, the assembler output listing file in Listing 5.16 is generated:

Listing 5.16 Generated listing file

```
 1    1                     YearAsString:  MACRO
 2    2                         DC.B $30+(\1 /1000)%10
```

```
 3   3                          DC.B $30+(\1 / 100)%10
 4   4                          DC.B $30+(\1 /  10)%10
 5   5                          DC.B $30+(\1 /   1)%10
 6   6                      ENDM
 7   7
 8   8           0000 0001  ifdef ADD_COPYRIGHT
 9   9                        ORG $1000
10  10   a001000 436F 7079    DC.B "Copyright by "
          001004 7269 6768
          001008 7420 6279
          00100C 20
11  11   a00100D 4A6F 686E    DC.B "John Doe"
          001011 2044 6F65
12  12           0000 0001    ifdef YEAR
13  13   a001015 2031 3939      DC.B " 2009-"
          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

Description

This option sets an environment variable.

Syntax

```
-Env<EnvironmentVariable>=<VariableSetting>
```

Group

Host

Scope

Assembly Unit

Arguments

`<EnvironmentVariable>`: Environment variable to be set

`<VariableSetting>`: Setting of the environment variable

Default

None

Example

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

This is the same as:

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

in the `default.env` file.

See also

Environment Variable Details

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

Description

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

Use the `-Fh` option to use a proprietary (HIWARE) object-file format.

With the `-F2` option set, the Assembler produces an ELF/DWARF object file. This object-file format may also be supported by other Compiler or Assembler vendors.

With the `-FA2` option set, the Assembler produces an ELF/DWARF absolute file. This file format may also be supported by other Compiler or Assembler vendors.

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

Syntax

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

Group

Output

Scope

Application

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`

Example

`ASMOPTIONS=-F2`

-H: Short Help

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.

Syntax

`-H`

Group

Various

Scope

None

Arguments

None

Default

None

Example

Listing 5.17 is a portion of the list produced by the `-H` option:

Listing 5.17 Example help list

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

-I: Include file path

Description

With the `-I` option it is possible to specify a file path used for include files.

Syntax

`-I<path>`

Group

Input

Scope

None

Arguments

`<path>`: File path to be used for includes

Default

None

Example

```
-Id:\mySources\include
```

-L: Generate a 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.

Syntax

```
-L[=<dest>]
```

Group

Output

Scope

Assembly unit

Arguments

`<dest>`: the name of the listing file to be generated.

It may contain special modifiers (see Using Special Modifiers).

Default

No generated listing file

Example

```
ASMOPTIONS=-L
```

In the following example of assembly code (Listing 5.18), 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 5.18, together with the code from an include file (Listing 5.19) generates the output listing in Listing 5.20.

Listing 5.18 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 5.19 Example source code from an include file

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

Listing 5.20   Assembly output listing

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

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

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

See also

Assembler options:

- -Lasmc: Configure listing file
- -Lasms: Configure the address size in the listing file
- -Lc: No macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: Not included file in listing file

-Lasmc: Configure listing file

Description

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

Syntax

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

Group

Output

Scope

Assembly unit

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.

Examples

For the following assembly source code, the Assembler generates the default-configured output listing (Listing 5.21):

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

Listing 5.21 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 5.22:

Listing 5.22 Example output listing

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

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

See also

Assembler options:

- -L: Generate a listing file
- -Lc: No macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: Not included file in listing file
- -Lasms: Configure the address size in the listing file

-Lasms: Configure the address size in the listing file

Description

The default-configured listing file shows a lot of information. With this option, the size of the address column can be reduced to the size of interest. To configure which columns are printed, see the -Lasmc: Configure listing file option. To configure which lines to print, see the following assembler options:

- -Lc: No macro call in listing file,
- -Ld: No macro definition in listing file,
- -Le: No macro expansion in listing file, and
- -Li: Not included file in listing file.

Syntax

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

Group

Output

Scope

Assembly unit

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`

Example

For the following instruction:

```
NOP
```

the Assembler generates this default-configured output listing (Listing 5.23):

Listing 5.23 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 5.24).

Listing 5.24 Example assembler output listing configured with -Lasms1

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

See also

Assembler Listing File

Assembler options:

- -Lasmc: Configure listing file
- -L: Generate a listing file
- -Lc: No macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: Not included file in listing file

-Lc: No macro call in listing file

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.

Syntax

`-Lc`

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

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 5.25, along with additional source code (Listing 5.26) from the `macro.inc` include file generates the output in the assembly listing file (Listing 5.27).

Listing 5.25 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 5.26 Example source code from the macro.inc file

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

Listing 5.27 Output assembly listing

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

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

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

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

See also

- -L: Generate a listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: Not included file in listing file

-Ld: No macro definition in listing file

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.

Syntax

`-Ld`

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

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 5.28 along with additional source code (Listing 5.29) from the `macro.inc` file generates an assembler output listing (Listing 5.30) file.

Listing 5.28 Example assembly source code

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

Listing 5.29 Example source code from an include file

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

Listing 5.30 Example assembler output listing

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

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

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

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

See also

- -L: Generate a listing file
- -Lc: No macro call in listing file
- -Le: No macro expansion in listing file
- -Li: Not included file in listing file

-Le: No macro expansion in listing file

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.

Syntax

```
-Le
```

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

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 5.31 along with additional source code (Listing 5.32) from the `macro.inc` file generates an assembly output listing file (Listing 5.33).

Listing 5.31 Example assembly source code

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

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

Listing 5.32 Example source code from an included file

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

Listing 5.33 Example assembler output listing

```
 Abs. Rel.   Loc    Obj. code  Source line
 ---- ----   ------ ---------  -----------
    1    1                              XDEF Start
    2    2                     MyData:  SECTION
    3    3   000000            char1:   DS.B  1
    4    4   000001            char2:   DS.B  1
    5    5                              INCLUDE "macro.inc"
    6    1i                    cpChar:  MACRO
    7    2i                             LDAA  \1
    8    3i                             STAA  \2
    9    4i                             ENDM
   10    6                     CodeSec: SECTION
   11    7                     Start:
   12    8                              cpChar char1, char2
   15    9   000006 A7                  NOP
```

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

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

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

See also

Assembler options:

- -L: Generate a listing file
- -Lc: No macro call in listing file
- -Ld: No macro definition in listing file
- -Li: Not included file in listing file

-Li: Not included file in listing file

Group

Output

Scope

Assembly unit

Syntax

`-Li`

Arguments

None

Default

None

Description

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

Example

`ASMOPTIONS=-Li`

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

When `-Li` option is specified, the assembly source code in Listing 5.34 along with additional source code (Listing 5.35) from the `macro.inc` file generates the following output in the assembly listing file (Listing 5.36).

Listing 5.34 Example assembly source code

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

Listing 5.35 Example source code in an include file

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

Listing 5.36 Example assembler output listing

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

The Assembler stores the macro definition, invocation, and expansion in the listing file. The Assembler does not store the content of included files in the listing file.

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

See also

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

-Lic: License information

Description

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

Syntax

`-Lic`

Group

Various

Scope

None

Arguments

None

Default

None

Example

```
ASMOPTIONS=-Lic
```

See also

Assembler options:

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

-LicA: License information about every feature in directory

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.

Syntax

`-LicA`

Group

Various

Scope

None

Arguments

None

Default

None

Example

`ASMOPTIONS=-LicA`

See also

Assembler options:

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

-LicBorrow: Borrow license feature

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.

Syntax

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

Group

Host

Scope

None

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-2009:18:35`).

Default

None

Defines

None

Pragmas

None

Example

```
-LicBorrowHI100100;3.000:12-Mar-2009:18:25
```

See also

Assembler options:

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

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

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.

Syntax

```
-LicWait
```

Group

Host

Scope

None

Arguments

None

Default

None

Example

```
ASMOPTIONS=-LicWait
```

See also

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

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

Description

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

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

Syntax

```
-M(s|b|l)
```

Group

Code Generation

Scope

Application

Arguments

`s`: small memory model

`b`: banked memory model

`l`: large memory model.

Default

```
-Ms
```

Example

```
ASMOPTIONS=-Ms
```

-MacroNest: Configure maximum macro nesting

Description

This option controls how deep macros calls can be nested. Its main purpose is to avoid endless recursive macro invocations. When the nesting level is reached, then the message A.

Syntax

```
-MacroNest<Value>
```

Group

Language

Scope

Assembly Unit

Arguments

`<Value>`: max. allowed nesting level

Default

`3000`

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

Description

This switches `ON` compatibility mode with the MCUasm Assembler. Additional features supported, when this option is activated are enumerated in MCUasm Compatibility.

Syntax

`-MCUasm`

Group

Various

Scope

Assembly Unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-MCUasm`

-N: Display notify box

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.

Syntax

`-N`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-N`

If an error occurs during assembling, an alert dialog box will be opened.

-NoBeep: No beep in case of an error

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.

Syntax

`-NoBeep`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-NoBeep
```

-NoDebugInfo: No debug information for ELF/DWARF files

Description

By default, the Assembler produces debugging info for the produced ELF/DWARF files. This can be switched off with this option.

Syntax

```
-NoDebugInfo
```

Group

Language

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-NoDebugInfo
```

-NoEnv: Do not use environment

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).

Syntax

```
-NoEnv
```

Group

Startup (This option cannot be specified interactively.)

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
xx.exe -NoEnv
```

(Use the actual executable name instead of "*xx*")

See also

Environment

-ObjN: Object filename specification

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.

Syntax

-ObjN<FileName>

Group

Output

Scope

Assembly Unit

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.

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

-Prod: Specify project file at startup

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.

Syntax

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

Group

None (This option cannot be specified interactively.)

Scope

None

Arguments

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

Default

None

Example

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

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

See also

Environment

-Struct: Support for structured types

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.

Syntax

```
-Struct
```

Group

Input

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-Struct
```

See also

Mixed C and Assembler Applications

-V: Prints the Assembler version

Description

Prints the Assembler version and the current directory.

NOTE Use this option to determine the current directory of the Assembler.

Syntax

```
-V
```

Group

Various

Scope

None

Arguments

None

Default

None

Example

-V produces the following listing (Listing 5.37):

Listing 5.37 Example of a version listing

```
Command Line '-v'
Assembler V-5.0.8, Jul  7 2009

Directory: C:\Freescale\demo

Common Module V-5.0.7, Date Jul  7 2009
User Interface Module, V-5.0.17, Date Jul  7 2009
Assembler Kernel, V-5.0.13, Date Jul  7 2009
Assembler Target, V-5.0.8, Date Jul  7 2009
```

-View: Application standard occurrence

Description

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

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

Syntax

`-View<kind>`

Group

Host

Scope

Assembly Unit

Arguments

`<kind>` is one of:

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

Default

Application is started with arguments: `Minimized`.

Application is started without arguments: `Window`.

Example

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

-W1: No information messages

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.

Syntax

```
-W1
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-W1
```

-W2: No information and warning messages

Description

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

Syntax

`-W2`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-W2`

-WErrFile: Create "err.log" error file

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.

Syntax

`-WErrFile(On|Off)`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

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

Example

- `-WErrFileOn`

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

- `-WErrFileOff`

  existing `err.log` is not modified.

See also

-WStdout: Write to standard output

-WOutFile: Create error listing file

-Wmsg8x3: Cut filenames in Microsoft format to 8.3

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.

Syntax

`-Wmsg8x3`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

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

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

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

See also

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

-WmsgCE: RGB color for error messages

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.

Syntax

```
-WmsgCE<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCE255` changes the error messages to blue.

-WmsgCF: RGB color for fatal messages

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.

Syntax

```
-WmsgCF<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCF255` changes the fatal messages to blue.

-WmsgCI: RGB color for information messages

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.

Syntax

`-WmsgCI<RGB>`

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCI255` changes the information messages to blue.

-WmsgCU: RGB color for user messages

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.

Syntax

`-WmsgCU<RGB>`

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCU255` changes the user messages to blue.

-WmsgCW: RGB color for warning messages

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.

Syntax

`-WmsgCW<RGB>`

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCW0` changes the warning messages to black.

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

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.

Syntax

`-WmsgFb[v|m]`

Group

Messages

Scope

Assembly Unit

Arguments

`v`: Verbose format

`m`: Microsoft format

Default

`-WmsgFbm`

Example

Assume that the assembly source code in Listing 5.38 is to be assembled in the batch mode.

Listing 5.38 Example assembly source code

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

The Assembler generates the error output (Listing 5.39) in the assembler window if it is running in batch mode:

Listing 5.39 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 5.40):

Listing 5.40 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

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]

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

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.

Syntax

`-WmsgFi[v|m]`

Group

Messages

Scope

Assembly Unit

Arguments

`v`: Verbose format

`m`: Microsoft format

Default

`-WmsgFiv`

Example

If the Assembler is running in interactive mode, the default error output is shown in the assembler window as in Listing 5.42.

Listing 5.41 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 5.42):

Listing 5.42 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

Assembler options:

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

-WmsgFob: Message format for batch mode

Description

With this option it is possible to modify the default message format in the batch mode. The formats in Listing 5.43 are supported (assumed that the source file is `x:\Freescale\sourcefile.asmx`).

Listing 5.43 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
```

Syntax

`-WmsgFob<string>`

Group

Messages

Scope

Assembly Unit

Arguments

`<string>`: format string (see Listing 5.43)

Default

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

Example

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

produces a message displayed in Listing 5.44 using the format. The options are set for producing the path of a file with its filename, extension, and line.

Listing 5.44 Message format in batch mode

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

See also

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

-WmsgFoi: Message format for interactive mode

Description

With this option it is possible modify the default message format in interactive mode. The formats in Listing 5.45 are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`):

Listing 5.45 Supported formats for the interactive mode

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

\n	New line

Syntax

```
-WmsgFoi<string>
```

Group

Messages

Scope

Assembly Unit

Arguments

<string>: format string (see Listing 5.45)

Default

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

Example

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

produces a message in following format (Listing 5.46):

Listing 5.46 Message format for interactive mode

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

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

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

-WmsgFonf: Message format for no file information

Description

Sometimes there is no file information available for a message (e.g., if a message not related to a specific file). Then this message format string is used. The formats in Listing 5.47 are supported:

Listing 5.47 Supported formats for the "no file information option"

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

Syntax

`-WmsgFonf<string>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

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

produces a message in following format:

`information L10324: Linking successful`

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

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

-WmsgFonp: Message format for no position information

Description

Sometimes there is no position information available for a message (e.g., if a message not related to a certain position). Then this message format string is used. The formats in Listing 5.48 are supported (supposed that the source file is `x:\Freescale\sourcefile.asmx`)

Listing 5.48 Supported formats for the “no position information” option

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

Syntax

```
-WmsgFonp<string>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

`ASMOPTIONS=-WmsgFonf”%k %d: %m\n”`

produces a message in following format:

`information L10324: Linking successful`

See also

ERRORFILE: Filename specification error environment variable

Assembler options:

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

-WmsgNe: Number of error messages

Description

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

Syntax

`-WmsgNe<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

`50`

Example

`ASMOPTIONS=-WmsgNe2`

The Assembler stops assembling after two error messages.

See also

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

-WmsgNi: Number of Information messages

Description

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

Syntax

`-WmsgNi<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

`50`

Example

```
ASMOPTIONS=-WmsgNi10
```

Only ten information messages are logged.

See also

Assembler options:

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

-WmsgNu: Disable user messages

Description

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

- `a`

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

- `b`

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

- `c`

  Disables messages informing about generated files.

- `d`

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

- `e`

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

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

Syntax

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

Group

Messages

Scope

None

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

Example

`-WmsgNu=c`

-WmsgNw: Number of warning messages

Description

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

Syntax

`-WmsgNw<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

`50`

Example

`ASMOPTIONS=-WmsgNw15`

Only 15 warning messages are logged.

See also

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

-WmsgSd: Setting a message to disable

Description

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

Syntax

`-WmsgSd<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

`-WmsgSd1801`

See also

- -WmsgSe: Setting a message to error
- -WmsgSi: Setting a message to information
- -WmsgSw: Setting a message to warning

-WmsgSe: Setting a message to error

Description

Allows changing a message to an error message.

Syntax

`-WmsgSe<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

`-WmsgSe1853`

See also

Assembler options:

- -WmsgSd: Setting a message to disable
- -WmsgSi: Setting a message to information
- -WmsgSw: Setting a message to warning

-WmsgSi: Setting a message to information

Description

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

Syntax

`-WmsgSi<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

`-WmsgSi1853`

See also

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to error
- -WmsgSw: Setting a message to warning

-WmsgSw: Setting a message to warning

Description

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

Syntax

`-WmsgSw<number>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

`-WmsgSw2901`

See also

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to error
- -WmsgSi: Setting a message to information

-WOutFile: Create error listing file

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.

Syntax

`-WOutFile(On|Off)`

Group

Messages

Scope

Assembly Unit

Arguments

None.

Default

Error listing file is created.

Example

`-WOutFileOn`

The error file is created as specified with `ERRORFILE`.

`-WErrFileOff`

No error file is created.

See also

- -WErrFile: Create "err.log" error file
- -WStdout: Write to standard output

-WStdout: Write to standard output

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`.

Syntax

`-WStdout(On|Off)`

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

Output is written to `stdout`

Example

`-WStdoutOn`

All messages are written to `stdout`.

`-WErrFileOff`

Nothing is written to `stdout`.

See also

- -WErrFile: Create "err.log" error file
- -WOutFile: Create error listing file

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 and Section Types. The last part of the chapter deals with the merits of using relocatable sections. (See Relocatable vs. Absolute Sections.)

Section Attributes

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

- Code Sections.
- Constant Sections, or
- Data Sections,

Code Sections

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

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

Constant Sections

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

Data Sections

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

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

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

Section Types

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

This section covers these two types of sections:

- Absolute Sections
- Relocatable Sections

Absolute Sections

The starting address of an absolute section is known at assembly time. An absolute section is defined through the ORG - Set location counter assembler directive. The operand specified in the ORG directive determines the start address of the absolute section. See Listing 6.1 for an example of constructing absolute sections using the ORG assembler directive.

Listing 6.1 Example source code using ORG for absolute sections

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

      ORG   $C00  ; Absolute code section.
```

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

In the previous example, two bytes of storage are allocated starting at address `$A00`. The *constant* variable - `cst1` - will be allocated one byte at address `$A00` and another constant - `cst2` - will be allocated one byte at address `$A01`. 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 `$A00` is not bigger than `$10` bytes, otherwise the section starting at `$A00` and the section starting at `$A10` 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.

Listing 6.2 Example PRM file for Listing 6.1

```
LINK  test.abs /* Name of the executable file generated.     */
NAMES test.o   /* Name of the object file in the application */
END
SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_ROM  = READ_ONLY  0x8000 TO 0xFDFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_RAM  = READ_WRITE 0x0100 TO 0x023F;
END

PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.          */
  DEFAULT_RAM, SSTACK INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM         INTO MY_ROM;
END
```

```
STACKSTOP $014F /* Initializes the stack pointer */
INIT   entry /* entry is the entry point to the application.   */
VECTOR ADDRESS 0xFFFE entry /* Initialization for Reset vector.*/
```

The linker PRM file contains at least:

- The name of the absolute file (`LINK` command).
- The name of the object file which should be linked (`NAMES` command).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined `DEFAULT_RAM` (or its ELF alias `'.data'`) section must be placed there. For applications containing only absolute sections, nothing will be allocated (`SECTIONS` and `PLACEMENT` commands).
- The specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section `DEFAULT_ROM` (or its ELF alias `'.data'`) must be placed there. For applications containing only absolute sections, nothing will be allocated (`SECTIONS` and `PLACEMENT` commands).
- The specification of the application entry point (`INIT` command)
- The definition of the reset vector (`VECTOR ADDRESS` command)

Relocatable Sections

The starting address of a relocatable section is evaluated at linking time according to the information stored in the linker parameter file. A relocatable section is defined through the SECTION - Declare relocatable section assembler directive. See Listing 6.3 for an example using the `SECTION` directive.

Listing 6.3 Example source code using SECTION for relocatable sections

```
          XDEF   entry
constSec: SECTION      ; Relocatable constant data section.
cst1:     DC.B   $A6
cst2:     DC.B   $BC
...
dataSec:  SECTION      ; Relocatable data section.
var:      DS.B   1

codeSec:  SECTION      ; Relocatable code section.
entry:
          LDAA   cst1  ; Loads value into cst1
          ADDA   cst2  ; Adds value in cst2
          STAA   var   ; Stores into var
          BRA    entry
```

In the previous example, two bytes of storage are allocated in the `constSec` section. The constant `cst1` is allocated at the start of the section at address `$A00` and another constant `cst2` is allocated at an offset of `1` byte from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable `constSec` section until another section is specified using the `ORG` or `SECTION` directives.

When using relocatable sections, the user does not need to care about overlapping sections. The linker will assign a start address to each section according to the input from the linker parameter file.

The user can decide to define only one memory area for the code and constant sections and another one for the variable sections or to split the sections over several memory areas.

Example: Defining one RAM and one ROM area.

When all constant and code sections as well as data sections can be allocated consecutively, the PRM file used to assemble the example above can be defined as in Listing 6.4.

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

```
LINK test.abs /* Name of the executable file generated.      */
NAMES
  test.o      /* Name of the object file in the application. */
END
SECTIONS
/* READ_ONLY memory area.   */
  MY_ROM = READ_ONLY  0x0B00 TO 0x0BFF;
/* READ_WRITE memory area. */
  MY_RAM = READ_WRITE 0x0800 TO 0x08FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.         */
  DEFAULT_RAM    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM    INTO MY_ROM;
END
INIT entry                  /* Application entry point.          */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

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

According to the PRM file above,

- the `dataSec` section will be allocated starting at `0x0800`.
- the `constSec` section will be allocated starting at `0x0B00`.
- the `codeSec` section will be allocated next to the `constSec` section.

Example: Defining Multiple RAM and ROM Areas

When all constant and code sections as well as data sections cannot be allocated consecutively, the PRM file used to link the example above can be defined as in Listing 6.5:

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

```
LINK test.abs  /* Name of the executable file generated.      */
NAMES
  test.o       /* Name of the object file in the application. */
END
SECTIONS
 /* Two READ_ONLY memory areas */
  ROM_AREA_1= READ_ONLY  0x8000 TO 0x800F;
  ROM_AREA_2= READ_ONLY  0x8010 TO 0xFDFF;
/* Three READ_WRITE memory areas */
  RAM_AREA_1= READ_WRITE 0x0040 TO 0x00FF; /* zero-page memory area */
  RAM_AREA_2= READ_WRITE 0x0100 TO 0x01FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec              INTO RAM_AREA_2;
```

```
  DEFAULT_RAM          INTO RAM_AREA_1;

/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec             INTO ROM_AREA_2;
  codeSec, DEFAULT_ROM INTO ROM_AREA_1;
END
INIT   entry                /* Application's entry point.         */
VECTOR 0 entry /* Initialization of the reset vector. */
```

The linker PRM file contains at least:

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

According to the PRM file in Listing 6.5,

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

Relocatable vs. Absolute Sections

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

Modularity

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

Multiple Developers

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

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

Early Development

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

Enhanced Portability

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

Tracking Overlaps

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

Reusability

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

7

Assembler Syntax

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

- Comment Line or a
- Source Line.

Comment Line

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

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

Listing 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,
- an Operation Field,
- one or several operands, or
- a comment.

Characters on the source line may be either upper or lower case. Directives and instructions are case-insensitive, whereas symbols are case-sensitive unless the assembler option for case insensitivity on label names (-Ci: Switch case sensitivity on label names OFF) is activated.

Label Field

The label field is the first field in a source line. A label is a symbol followed by a colon. Labels can include letters (A–Z or a–z), underscores, periods, and numbers. The first character must not be a number.

NOTE For compatibility with other macro assembler vendors, an identifier starting on column 1 is considered to be a label, even when it is not terminated by a colon. When the -MCUasm: Switch compatibility with MCUasm ON assembler option is activated, you *MUST* terminate labels with a colon. The Assembler produces an error message when a label is not followed by a colon.

Labels are required on assembler directives that define the value of a symbol (`SET` or `EQU`). For these directives, labels are assigned the value corresponding to the expression in the operand field.

Labels specified in front of another directive, instruction or comment are assigned the value of the location counter in the current section.

NOTE When the Macro Assembler expands a macro it generates internal symbols starting with an underscore `'_'`. Therefore, to avoid potential conflicts, user defined symbols should not begin with an underscore

NOTE For the Macro Assembler, a `.B` or `.W` at the end of a label has a specific meaning. Therefore, to avoid potential conflicts, user- defined symbols should not end with `.B` or `.W`.

Operation Field

The operation field follows the label field and is separated from it by a white space. The operation field must not begin in the first column. An entry in the operation field is one of the following:

- an instruction's mnemonic - an abbreviated, case-insensitive name for a member in the Instruction Set
- a Directive name, or
- a Macro name.

Instruction Set

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

Table 7.1 presents an overview of the instructions available:

Table 7.1 HC12, HCS12, and HCS12X Instruction Set

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Instruction	Description
NEGA	2’s complement accumulator A
NEGB	2’s complement accumulator B
NEGW	2’s complement in memory
NEGX	2’s complement register X
NEGY	2’s complement register Y
NOP	No operation
ORAA	Logical OR with Accumulator A
ORAB	Logical OR with Accumulator B
ORCC	Logical OR with CCR
ORX	Logical OR register X with memory
ORY	Logical OR register Y with memory
PSHA	Push register A
PSHB	Push register B
PSHC	Push register CCR
PSHCW	Push register CCRW
PSHD	Push register D
PSHX	Push register X
PSHY	Push register Y
PULA	Pop register A
PULB	Pop register B
PULC	Pop register CCR
PULCW	Pop register CCRW
PULD	Pop register D
PULX	Pop register X
PULY	Pop register Y

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

Instruction	Description
REV	MIN-MAX Rule Evaluation for 8-bit values
REVW	MIN-MAX Rule Evaluation for 16-bit values
ROL	Rotate memory left
ROLA	Rotate accumulator A left
ROLB	Rotate accumulator B left
ROLW	Rotate memory left
ROLX	Rotate register X left
ROLY	Rotate register Y left
ROR	Rotate memory right
RORA	Rotate accumulator A Right
RORB	Rotate accumulator B Right
RORW	Rotate memory Right
RORX	Rotate register X Right
RORY	Rotate register Y Right
RTC	Return from CALL
RTI	Return from Interrupt
RTS	Return from Subroutine
SBA	Subtract accumulator A and B
SBCA	Subtract with Carry from accumulator A
SBCB	Subtract with Carry from accumulator B
SBED	Subtract with borrow from accumulator D
SBEX	Subtract with borrow from register X
SBEY	Subtract with borrow from register Y
SEC	Set carry bit
SEI	Set interrupt bit

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

Instruction	Description
SEV	Set two’s complement overflow bit
SEX	Sign Extend into 16-bit register
STAA	Store Accumulator A
STAB	Store Accumulator B
STD	Store Accumulator D
STOP	Stop
STS	Store register SP
STX	Store register X
STY	Store register Y
SUBA	Subtract without carry from accumulator A
SUBB	Subtract without carry from accumulator B
SUBD	Subtract without carry from accumulator D
SUBX	Subtract without carry from register X
SUBY	Subtract without carry from register Y
SWI	Software interrupt
TAB	Transfer A to B
TAP	Transfer A to CCR
TBA	Transfer B to A
TBEQ	Test counter and branch if zero
TBL	8-bit Table Lookup and Interpolate
TBNE	Test counter and branch if Not zero
TFR	Transfer Register to register
TPA	Transfer CCR to A
TRAP	Software Interrupt
TST	Test memory for 0 or minus

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

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

Directive

Assembler directives are described in the Assembler Directives 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

The operand fields, when present, follow the operation field and are separated from it by a white space. When two or more operand subfields appear within a statement, a comma must separate them.

The addressing mode notations in Table 7.2 are allowed in the operand field:

Table 7.2 S12(X) Addressing Modes

Addressing Mode	Notation
Inherent	No operands
Immediate	#<immediate 8-bit expression> or #<immediate 16-bit expression>
Direct	<8-bit address>
Extended	<16-bit address>
Relative	<PC relative, 8-bit offset> or <PC relative, 16-bit offset>
Indexed, 5-bit Offset	<5-bit offset>, xysp
Indexed, 9-bit Offset	<9-bit offset>, xysp
Indexed, 16-bit Offset	<16-bit offset>, xysp
Indexed, Indirect 16-bit Offset	[<16-bit offset>, xysp]
Indexed, Pre-Decrement	<3-bit offset>, -xys
Indexed, Pre-Increment	<3-bit offset>, +xys
Indexed, Post-Decrement	<3-bit offset>, xys-
Indexed, Post-Increment	<3-bit offset>, xys+
Indexed, Accumulator Offset	abd, xysp
Indexed-Indirect, Accumulator D Offset	[D, xysp]
Global	New instructions beginning with the label G are created for this usage, such as GLDAA, GSTAA, etc.

In Table 7.2:

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

Inherent

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

Listing 7.2 Inherent addressing-mode instructions

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

Immediate

The opcode contains the value to use with the instruction rather than the address of this value. The `'#'` character is used to indicate an immediate addressing mode operand (Listing 7.3).

Listing 7.3 Immediate addressing mode

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

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

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

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

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

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

Omitting the # character causes the Assembler to misinterpret the expression as an address rather than an explicit data (Listing 7.5).

Listing 7.5 Potential error - direct addressing mode instead of immediate

```
            LDAA   $60
```

The code above means to load accumulator `A` with the value stored at address `$60`.

Direct

On the S12(X), the direct addressing mode is used to address operands in the direct page of the memory (location `$0000 to $00FF`, also called the zero page). On the HCS12X the direct page register (`DIRECT`) determines the position of the direct page within the memory map.The direct-addressing mode is used to access operands in the address range `$00` through `$FF` in the direct page. Accesses on this memory range are faster and require less code than using the extended addressing mode (see Listing 7.6). To speed up an application a programmer can decide to place the most commonly accessed data in this area of memory.

Listing 7.6 Direct addressing mode in an absolute section

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

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

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

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

Listing 7.7 Direct addressing modes in a relocatable section

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

Extended

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

Listing 7.8 Extended addressing mode

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

Relative

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

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

Listing 7.9 Relative addressing mode

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

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

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

Listing 7.10 Using BRA with an offset

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

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

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

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

- labels defined in any absolute section
- labels defined in any relocatable section
- external labels (defined in an `XREF` directive)

Indexed, 5-bit Offset

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

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

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

Listing 7.11 Indexed (5-bit offset) addressing mode

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

            LDY #DATA_TBL
            STAA 8, Y
```

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

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

Indexed, 9-bit Offset

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

For information about Indexed-`PC` and Indexed-`PC`-Relative addressing modes, see Indexed-PC vs. Indexed-PC Relative Addressing Mode.

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

Listing 7.12 Indexed, 9-bit offset addressing mode

```
          ORG $1000
CST_TBL:  DC.B  $5,  $10, $18, $20, $28, $30, $38, $40, $48
          DC.B  $50, $58, $60, $68, $70, $78, $80, $88, $90
```

```
          DC.B   $98, $A0, $A8, $B0, $B8, $C0, $C8, $D0, $D8
          ORG    $800
DATA_TBL: DS.B   40
main:
          LDX    #CST_TBL
          LDAA   20,X

          LDY    #DATA_TBL
          STAA   18, Y
```

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

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

Indexed, 16-bit Offset

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

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

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

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

Listing 7.13 Indexed, 16-bit offset addressing mode

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

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

Indexed, Indirect 16-bit Offset

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

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

In Listing 7.14, the offset 4 is added to the value of register X ($1000) to form the address $1004. Then an address pointer ($2001) is read from memory at $1004. The accumulator A is loaded with $35, the value stored at address $2001.

Listing 7.14 Indexed, indirect 16-bit offset addressing mode

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

Indexed, Pre-Decrement

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

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

Listing 7.15 Indexed, pre-decrement addressing mode

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

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

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

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

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

Indexed, Pre-Increment

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

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

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

Listing 7.16 Indexed, pre-increment addressing mode

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

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

Indexed, Post-Decrement

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

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

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

Listing 7.17 Indexed, post-increment addressing mode

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

Indexed, Post-Increment

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

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

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

Listing 7.18 Indexed, post-increment addressing mode

```
            ORG $1000

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

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

Indexed, Accumulator Offset

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

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

Listing 7.19 Indexed, accumulator offset addressing mode

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

Indexed-Indirect, Accumulator D Offset

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

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

Listing 7.20 Index-indirect, accumulator D offset addressing mode

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

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

Global

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

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

Listing 7.21 Global addressing mode

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

Indexed-PC vs. Indexed-PC Relative Addressing Mode

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

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

Listing 7.22 Using the indexed-PC addressing mode

```
main:
            LDAB  3, PC
            DC.B  $20, $30, $40, $50
```

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

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

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

Listing 7.23 Using the indexed-PC relative addressing mode

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

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

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

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

- labels defined in any absolute section
- labels defined in any relocatable section
- external labels (defined in an `XREF` directive)

Comment Field

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

Listing 7.24 Example of a comment

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

Symbols

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

- User-Defined symbols
- External Symbols
- Undefined Symbols
- Reserved Symbols

User-Defined symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

- The section, in which the memory location is defined
- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION - Declare relocatable section assembler directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line (Listing 7.25).

Listing 7.25 Example of a user-defined relocatable SECTION

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec.
label2: DC.B 5 ; label2 is assigned offset 2 within Sec.
label3: DC.B 1 ; label3 is assigned offset 7 within Sec.
```

It is also possible to define a label with either an absolute or a previously defined relocatable value, using the SET - Set symbol value or EQU - Equate symbol value assembler directives.

Symbols with absolute values must be defined with constant expressions.

Listing 7.26 Example of a user-defined absolute and relocatable SECTION

```
Sec: SECTION
label1: DC.B 2     ; label1 is assigned offset 0 within Sec.
label2: EQU  5     ; label2 is assigned value 5.
label3: EQU label1 ; label3 is assigned the address of label1.
```

External Symbols

A symbol may be made external using the XDEF - External symbol definition 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 7.27 for an example of using `XDEF` and `XREF`.

Listing 7.27 Examples of external symbols

```
      XREF extLabel      ; symbol defined in an other module.
                         ; extLabel is imported in the current module
      XDEF label         ; symbol is made external for other modules
                         ; label is exported from the current module
constSec: SECTION
label:    DC.W  1, extLabel
```

Undefined Symbols

If a label is neither defined in the source file nor declared external using `XREF`, the Assembler considers it to be undefined and generates an error message. Listing 7.28 shows an example of an undeclared label.

Listing 7.28 Example of an undeclared label

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

Reserved Symbols

Reserved symbols cannot be used for user-defined symbols.

Register names are reserved identifiers.

For the HC12 processor these reserved identifiers are:

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

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

Constants

The Assembler supports integer and ASCII string constants:

Integer Constants

The Assembler supports four representations of integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9).

  Example: 5, 512, 1024

- A hexadecimal constant is defined by a dollar character (`$`) followed by a sequence of hexadecimal digits (0-9, a-f, A-F).

  Example: `$5`, `$200`, `$400`

- An octal constant is defined by the commercial at character (`@`) followed by a sequence of octal digits (0-7).

  Example: `@5`, `@1000`, `@2000`

- A binary constant is defined by a percent character followed by a sequence of binary digits (0-1)

  Example: `%101`, `%1000000000`, `%10000000000`

The default base for integer constant is initially decimal, but it can be changed using the BASE - Set number base assembler directive. When the default base is not decimal, decimal values cannot be represented, because they do not have a prefix character.

String Constants

A string constant is a series of printable characters enclosed in single (`'`) or double quote (`"`). Double quotes are only allowed within strings delimited by single quotes. Single quotes are only allowed within strings delimited by double quotes. See Listing 7.29 for a variety of string constants.

Listing 7.29 String constants

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

Floating-Point Constants

The Macro Assembler does not support floating-point constants.

Operators

Operators recognized by the Assembler in expressions are:

- Addition and subtraction operators (binary)
- Multiplication, division and modulo operators (binary)
- Sign operators (unary)
- Shift operators (binary)
- Bitwise operators (binary)
- Logical operators (unary)
- Relational operators (binary)
- HIGH operator
- PAGE operator
- Force operator (unary)

Addition and subtraction operators (binary)

Description

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

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.

Syntax

```
Addition:    <operand> + <operand>
Subtraction: <operand> - <operand>
```

Example

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

Listing 7.30 Addition and subtraction operators

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

Multiplication, division and modulo operators (binary)

Description

The multiplication, division, and modulo operators are `*`, `/`, and `%`, respectively.

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.

Syntax

```
Multiplication: <operand> * <operand>
Division:       <operand> / <operand>
Modulo:         <operand> % <operand>
```

Example

See Listing 7.31 for an example of the multiplication, division, and modulo operators.

Listing 7.31 Multiplication, division, and modulo operators

```
23 * 4    ; multiplication (= 92)
23 / 4    ; division (= 5)
23 % 4    ; remainder(= 3)
```

Sign operators (unary)

Description

The (unary) sign operators are + and `-`.

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.

Syntax

`Plus:  +<operand>`

`Minus: -<operand>`

Example

See Listing 7.32 for an example of the unary sign operators.

Listing 7.32 Unary sign operators

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

Shift operators (binary)

Description

The binary shift operators are << and >>.

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.

Syntax

`Shift left:  <operand> << <count>`

`Shift right: <operand> >> <count>`

Example

See Listing 7.33 for an example of the binary shift operators.

Listing 7.33 Binary shift operators

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

Bitwise operators (binary)

Description

The binary bitwise operators are &, |, and ^.

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.

Syntax

```
Bitwise AND:       <operand> & <operand>
Bitwise OR:        <operand> | <operand>
Bitwise XOR:       <operand> ^ <operand>
```

Example

See Listing 7.34 for an example of the binary bitwise operators

Listing 7.34 Binary bitwise operators

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

Bitwise operators (unary)

Description

The unary bitwise operator is ~.

The ~ operator evaluates the one’s complement of the operand.

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

Syntax

`One’s complement: ~<operand>`

Example

See Listing 7.35 for an example of the unary bitwise operator.

Listing 7.35 Unary bitwise operator

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

Logical operators (unary)

Description

The unary logical operator is `!`.

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.

Syntax

`Logical NOT: !<operand>`

Example

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

Listing 7.36 Unary logical operator

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

Relational operators (binary)

Description

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

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.

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>
```

Example

See Listing 7.37 for an example of the binary relational operators

Listing 7.37 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

Description

The HIGH operator is `HIGH`.

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

Syntax

```
High Byte: HIGH(<operand>)
```

Example

Assume `data1` is a word located at address `$1050` in the memory.

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

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

```
LDAA  HIGH(data1)
```

This instruction will load the direct value at memory location of the higher byte of the address of `data1` (i.e., the value in memory location `$10`) in register A.

LOW operator

Description

The LOW operator is `LOW`.

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

Syntax

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

Example

Assume `data1` is a word located at address `$1050` in the memory.

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

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

```
LDAA  LOW(data1)
```

This instruction will load the direct value at memory location of the lower byte of the address of `data1` (i.e., the value in memory location `$50`) in register A.

PAGE operator

Description

The PAGE operator is `PAGE`.

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

Syntax

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

Example

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

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

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

```
LDAA  PAGE(data1)
```

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

Force operator (unary)

Description

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

The < or `.B` operators force direct addressing, whereas the > or `.W` operators force extended addressing.

Use the < operator to force 8-bit indexed or 8-bit direct addressing mode for an instruction.

Use the > operator to force 16-bit indexed or 16-bit extended addressing mode for an instruction.

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

Syntax

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

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

Example

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

Operator Precedence

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

Table 7.3 Operator precedence priorities

Operator	Description	Associativity
()	Parenthesis	Right to Left
~ + -	One's complement Unary Plus Unary minus	Left to Right
* / %	Integer multiplication Integer division Integer modulo	Left to Right
+ -	Integer addition Integer subtraction	Left to Right
<< >>	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

Table 7.3 Operator precedence priorities (*continued*)

Operator	Description	Associativity
^	Bitwise Exclusive OR	Left to Right
|	Bitwise OR	Left to Right

Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols
- External symbols
- The special symbol '*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the `DC` directive:

  ```
  DC.W  1, 2, *-2
  ```

Once a valid expression has been fully evaluated by the Assembler, it is reduced as one of the following type of expressions:

- Absolute Expression: The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant. Simple Relocatable Expression: The expression evaluates to an absolute offset from the start of a single relocatable section.
- Complex relocatable expression: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The Assembler does not support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in `XREF` directives, which are assumed to be relocatable symbols.

Absolute Expression

An absolute expression is an expression involving constants or known absolute labels or expressions. An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

See Listing 7.38 for an example of an absolute expression.

Listing 7.38 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 7.39 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 7.40):

Listing 7.40 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 7.41, 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 7.41 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 7.42 Example of relocatable expression

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

Unary Operation Result

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

Table 7.4 Expression type resulting from operator and operand type

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

Binary Operations Result

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

Table 7.5 Expression type resulting from operator and their operands

Operator	Left Operand	Right Operand	Expression
-	absolute	absolute	absolute
-	relocatable	absolute	relocatable
-	absolute	relocatable	complex
-	relocatable	relocatable	absolute
+	absolute	absolute	absolute
+	relocatable	absolute	relocatable
+	absolute	relocatable	relocatable
+	relocatable	relocatable	complex
*, /, %, <<, >>, |, &, ^	absolute	absolute	absolute
*, /, %, <<, >>, |, &, ^	relocatable	absolute	complex
*, /, %, <<, >>, |, &, ^	absolute	relocatable	complex
*, /, %, <<, >>, |, &, ^	relocatable	relocatable	complex

Translation Limits

The following limitations apply to the Macro Assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Includes may be nested up to `50`.
- The maximum line length is `1023`.

8

Assembler Directives

There are different classes of assembler directives. The following tables give an overview of the different directives and their classes.

Directive Overview

Section Definition Directives

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

Table 8.1 Directives for Defining Sections

Directive	Description
ORG - Set location counter	Define an absolute section
SECTION - Declare relocatable section	Define a relocatable section
OFFSET - Create absolute symbols	Define an offset section

Constant Definition Directives

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

Table 8.2 Directives for Defining Constants

Directive	Description
EQU - Equate symbol value	Assign a name to an expression (cannot be redefined)
SET - Set symbol value	Assign a name to an expression (can be redefined)

Data Allocation Directives

The directives in Table 8.3 are used to allocate variables.

Table 8.3 Directives for Allocating Variables

Directive	Description
DC - Define Constant	Define a constant variable
DCB - Define constant block	Define a constant block
DS - Define space	Define storage for a variable
RAD50 - Rad50-encoded string constants	RAD50 encoded string constants

Symbol Linkage Directives

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

Table 8.4 Symbol Linkage Directives

Directive	Description
ABSENTRY - Application entry point	Specify the application entry point when an absolute file is generated
XDEF - External symbol definition	Make a symbol public (visible from outside)
XREF - External symbol reference	Import reference to an external symbol.
XREFB - External reference for symbols located on the direct page	Import reference to an external symbol located on the direct page.

Assembly Control Directives

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

Table 8.5 Assembly control Directives

Directive	Description
ALIGN - Align location counter	Define Alignment Constraint
BASE - Set number base	Specify default base for constant definition

Table 8.5 Assembly control Directives (*continued*)

Directive	Description
END - End assembly	End of assembly unit
ENDFOR - End of FOR block	End of FOR block
EVEN - Force word alignment	Define 2-byte alignment constraint
FAIL - Generate error message	Generate user defined error or warning messages
FOR - Repeat assembly block	Repeat assembly blocks
INCLUDE - Include text from another file	Include text from another file.
LONGEVEN - Forcing long-word alignment	Define 4 Byte alignment constraint

Listing File Control Directives

Listing file control directives (Table 8.6) control the generation of the assembler listing file.

Table 8.6 Listing File Control Directives

Directive	Description
CLIST - List conditional assembly	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable listing	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set line length	Define line length in assembly listing file.
MLIST - List macro expansions	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable listing	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable paging	Disable paging in the assembly listing file.
PAGE - Insert page break	Insert page break.

Table 8.6 Listing File Control Directives (*continued*)

Directive	Description
PLEN - Set page length	Define page length in the assembler listing file.
SPC - Insert blank lines	Insert an empty line in the assembly listing file.
TABS - Set tab length	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide listing title	Define the user defined title for the assembler listing file.

Macro Control Directives

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

Table 8.7 Macro Control Directives

Directive	Description
ENDM - End macro definition	End of user defined macro.
MACRO - Begin macro definition	Start of user defined macro.
MEXIT - Terminate macro expansion	Exit from macro expansion.

Conditional Assembly Directives

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

Table 8.8 Conditional Assembly Directives

Directive	Description
ELSE - Conditional assembly	Alternate block
-Compat: Compatibility modes assembler option	End of conditional block
IF - Conditional assembly	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly	Test whether two string expressions are equal.

Table 8.8 Conditional Assembly Directives (*continued*)

Directive	Description
IFDEF	Test whether a symbol is defined.
IFEQ	Test whether an expression is null.
IFGE	Test whether an expression is greater or equal to 0.
IFGT	Test whether an expression is greater than 0.
IFLE	Test whether an expression is less or equal to 0.
IFLT	Test whether an expression is less than 0.
IFNC	Test whether two string expressions are different.
IFNDEF	Test whether a symbol is undefined
IFNE	Test whether 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

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.

Syntax

```
ABSENTRY <label>
```

Synonym

None

Example

If the example in Listing 8.1 is assembled using the -FA2 assembler option, an ELF/DWARF 2.0 Absolute file is generated.

Listing 8.1 Using ABSENTRY to specify an application entry point

```
       ABSENTRY entry

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

According to the `ABSENTRY` directive, the entry point will be set to the address of entry in the header of the absolute file.

ALIGN - Align location counter

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.

Syntax

`ALIGN <n>`

Synonym

None

Example

The example shown in Listing 8.2 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

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.

Syntax

`BASE <n>`

Synonym

None

Example

See Listing 8.3 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
```

WARNING! 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

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`.

Syntax

`CLIST [ON|OFF]`

Synonym

None

Example

Listing 8.4 is an example using the `CLIST OFF` option.

Listing 8.4 Listing file with CLIST OFF

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

Listing 8.5 is the corresponding listing file.

Listing 8.5 Example assembler listing using CLIST OFF

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

Listing 8.6 is a listing file using `CLIST ON`.

Listing 8.6 CLIST ON is selected

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

```
        ELSE
          LDAA #0
        ENDIF
```

Listing 8.7 is the corresponding listing file.

Listing 8.7 Example assembler listing using CLIST ON

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

DC - Define Constant

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.

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)`

Examples

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

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

- DCB - Define constant block
- DS - Define space
- ORG - Set location counter
- SECTION - Declare relocatable section

DCB - Define constant block

Description

The DCB directive causes the Assembler to allocate a memory block initialized with the specified <value>. The length of the block is the product: <size>*<count>.

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

The value of each storage unit allocated is the sign-extended expression <value>, which may contain forward references. The <count> cannot be relocatable. This directive does not perform any alignment.

Syntax

```
[<label>:] DCB [.<size>] <count>, <value>
```

where `<size> = B (default), W, or L`.

Examples

The following rules apply to size specifications for DCB directives (Listing 8.11):

- DCB.B: One byte is allocated for numeric expressions.
- DCB.W: Two bytes are allocated for numeric expressions.
- DCB.L: Four bytes are allocated for numeric expressions.

Listing 8.11 Assembly output listing showing the allocation of constants

```
000000 FFFF FF     Label: DCB.B 3, $FF
000003 FFFE FFFE          DCB.W 3, $FFFE
000007 FFFE
000009 0000 FFFE          DCB.L 3, $FFFE
00000D 0000 FFFE
000011 0000 FFFE
```

See also

- DC - Define Constant
- DS - Define space
- ORG - Set location counter
- SECTION - Declare relocatable section

DS - Define space

Description

The `DS` directive is used to reserve memory for variables (Listing 8.12). The content of the memory reserved is not initialized. The length of the block is the product:

`<size> * <count>`.

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

Listing 8.12 Examples of DS directives

```
Counter:  DS.B  2 ; 2 continuous bytes in memory
          DS.B  2 ; 2 continuous bytes in memory
                  ; can only be accessed through the label Counter
          DS.W  5 ; 5 continuous words in memory
```

The label `Counter` references the lowest address of the defined storage area.

NOTE Storage allocated with a `DS` directive may end up in constant data section or even in a code section, if the same section contains constants or code as well. The Assembler allocates only a complete section at once.

Syntax

`[<label>:] DS[.<size>] <count>`

where `<size>` = `B` (default), `W`, or `L`.

Synonym

`RMB (= DS.B)`

`RMD (2 bytes)`

`RMQ (4 bytes)`

Example

In Listing 8.13, a variable, a constant, and code were put in the same section. Because code has to be in ROM, then all three elements must be put into ROM. In order to allocate them separately, put them in different sections (Listing 8.14).

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 Proper memory allocation

```
DataSect:       SECTION    ; separate section for variables
Counter:        DS 1       ; 1-byte used

ConstSect:      SECTION    ; separate section for constants
InitialCounter: DC.B $f5   ; constant $f5

CodeSect:       SECTION    ; section for code
main:            NOP       ; NOP instruction
```

An `ORG` directive also starts a new section.

See also

- DC - Define Constant
- ORG - Set location counter
- SECTION - Declare relocatable section

ELSE - Conditional assembly

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.

Syntax

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

```
[ELSE]
  [<Block 2 - assembly language statements>]
ENDIF
```

Synonym

```
ELSEC
```

Example

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

Listing 8.15 Various conditional assembly directives

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

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

Listing 8.16 Output listing of Listing 8.15

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

END - End assembly

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.

Syntax

`END`

Synonym

`None`

Example

The `END` statement in Listing 8.17 causes any source code after the END statement to be ignored, as in Listing 8.18.

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

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.

Syntax

```
ENDFOR
```

Synonym

None

Example

See Listing 8.28 in the `FOR`.section.

See also

FOR - Repeat assembly block assembler directive

-Compat: Compatibility modes assembler option

ENDIF - End conditional assembly

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.

Syntax

```
ENDIF
```

Synonym

```
ENDC
```

Example

See Listing 8.30 in the `IF` section.

See also

IF - Conditional assembly assembler directive

ENDM - End macro definition

Description

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

Syntax

`ENDM`

Synonym

None

Example

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

Listing 8.19 Using ENDM to terminate a macro definition

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

EQU - Equate symbol value

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.

Syntax

```
<label>: EQU <expression>
```

Synonym

None

Example

See Listing 8.20 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

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.

Syntax

`EVEN`

Synonym

None

Example

See Listing 8.21 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 assembly directive

FAIL - Generate error message

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.

Syntax

```
FAIL <arg>|<string>
```

Synonym

None

Examples

The assembly code in Listing 8.22 generates the error messages in Listing 8.23. The value of the operand associated with the ‘`FAIL  200`’ or ‘`FAIL  600`’ directives determines (1) the format of any warning or error message and (2) whether the source code segment will be assembled.

Listing 8.22 Example source code

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

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

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

```
>> 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 is another assembly code example which again incorporates the `FAIL 200` and the `FAIL 600` directives. Listing 8.25 is the error message that was generated as a result of assembling the source code in Listing 8.24.

Listing 8.24 Example source code

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

          IFC "\2", ""
            FAIL  600
          ELSE
            STAA  \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
         cpChar, char2
```

Listing 8.25 Error messages resulting from assembling the source code in Listing 8.24

```
>> 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 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.

Listing 8.26 Example source code

```
cpChar: MACRO
          IFC "\1", ""
            FAIL  "A character must be specified as first parameter"
            MEXIT
          ELSE
            LDAA  \1
          ENDIF

          IFC "\2", ""
            FAIL  600
          ELSE
            STAA  \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
         cpChar, char2
```

Listing 8.27 Error messages resulting from assembling the source code in Listing 8.26

```
>> 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

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.

Syntax

```
FOR <label>=<num> TO <num>
ENDFOR
```

Synonym

None

Example

Listing 8.28 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

ENDFOR - End of FOR block

-Compat: Compatibility modes assembler option

IF - Conditional assembly

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).

Syntax

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

Synonym

`None`

Example

Listing 8.30 is an example of the use of conditional assembly directives

Listing 8.30 IF and ENDIF

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

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

Listing 8.31 Output listing after conditional assembly

```
    1    1           0000 0000   Try: EQU    0
    2    2           0000 0000        IF Try != 0
    4    4                            ELSE
    4    4    000000 8667               LDAA   #103
    6    6                            ENDIF
```

IFcc - Conditional assembly

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 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>
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Syntax

```
IFcc <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

None

Example

In Listing 8.32 the value of `Try` determines the instruction to be assembled in the program. As shown, the `ldaa #0` instruction is assembled. Changing the directive to `IFEQ` causes the `ldaa #103` instruction to be assembled instead.

Listing 8.32 shows the use of conditional assembler directives.

Listing 8.32 Using the IFNE conditional assembler directive

```
Try: EQU   0
     IFNE Try
```

```
    LDAA   #103
  ELSE
    LDAA   #0
  ENDIF
```

The value of `Try` determines the instruction to be assembled in the program. As shown, the `ldaa    #0` instruction is assembled. Changing the directive to `IFEQ` causes the `ldaa    #103` instruction to be assembled instead.

Listing 8.33 shows the listing provided by the Assembler for these lines of code.

Listing 8.33 Output listing for Listing 8.32

```
    1    1          0000 0000   Try: EQU   0
    2    2          0000 0000        IFNE Try
    4    4                           ELSE
    5    5   000000 8600               LDAA   #0
    6    6                           ENDIF
```

INCLUDE - Include text from another file

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.

Syntax

```
INCLUDE <file specification>
```

Synonym

None

Example

```
INCLUDE "..\LIBRARY\macros.inc"
```

LIST - Enable listing

Description

Specifies that instructions following this directive must be inserted into the listing and into the debug file. This is a default option. The listing file is only generated if the -L: Generate a listing file assembler option is specified on the command line.

The source text following the `LIST` directive is listed until a NOLIST - Disable listing or an END - End assembly assembler directive is reached

This directive is not written to the listing and debug files.

Syntax

```
LIST
```

Synonym

None

Example

The assembly source code using the `LIST` and `NOLIST` directives in Listing 8.34 generates the output listing in Listing 8.35.

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

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

```
   1    1   000000 A7           aaa:    NOP
   2    2
   4    4   000001 A7           bbb:    NOP
   5    5   000002 A7                   NOP
   6    6
  12   12   000005 A7           ddd:    NOP
  13   13   000006 A7                   NOP
```

LLEN - Set line length

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.

Syntax

```
LLEN <n>
```

Synonym

None

Example

The following portion of code in Listing 8.36 generates the listing file in Listing 8.37. 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

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.

Syntax

`LONGEVEN`

Synonym

None

Example

See Listing 8.38 for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

Listing 8.38 Forcing Long Word Alignment

```
   2    2   000000 01                  dcb.b 1,1
             ; location counter is not a multiple of 4; three filling
             ; bytes are required.
   3    3   000001 0000 00             longeven
   4    4   000004 0002 0002           dcb.w 2,2
             ; location counter is already a multiple of 4; no filling
             ; bytes are required.
   5    5                              longeven
   6    6   000008 0202                dcb.b 2,2
   7    7    ; following is for text section
   8    8                  s27         SECTION 27
```

```
    9    9   000000 9D                         nop
              ; location counter is not a multiple of 4; three filling
              ; bytes are required.
   10   10   000001 0000 00                    longeven
   11   11   000004 9D                         nop
```

See Also

ALIGN - Align location counter assembler directive

MACRO - Begin macro definition

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.

Syntax

```
<label>: MACRO
```

Synonym

None

Example

See Listing 8.39 for a macro definition.

Listing 8.39 Example macro definition

```
          XDEF  Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
cpChar:  MACRO
           LDAA  \1
           STAA  \2
         ENDM
CodeSec: SECTION
Start:
         cpChar char1, char2
```

MEXIT - Terminate macro expansion

Description

`MEXIT` is usually used together with conditional assembly within a macro. In that case it may happen that the macro expansion should terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the ENDM - End macro definition directive.

Syntax

```
MEXIT
```

Synonym

None

Example

See Listing 8.40 allows the replication of simple instructions or directives using `MACRO` with `MEXIT`.

Listing 8.40 Example assembly code using MEXIT

```
         XDEF  entry
storage: EQU   $00FF

save:    MACRO            ; Start macro definition
           LDX   #storage
           LDAA  \1
           STAA  0,x      ; Save first argument
           LDAA  \2
           STAA  2,x      ; Save second argument
           IFC   '\3', '' ; Is there a third argument?
             MEXIT        ; No, exit from macro.
             ENDC
           LDAA  \3       ; Save third argument
           STAA  4,X
         ENDM             ; End of macro definition

datSec: SECTION
char1:  ds.b  1
char2:  ds.b  1

codSec: SECTION
entry:
```

```
        save  char1, char2
```

Listing 8.41 shows the macro expansion of the previous macro.

Listing 8.41 Macro expansion of Listing 8.40

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF   entry
    2    2
    3    3          0000 00FF   storage: EQU    $00FF
    4    4
    5    5                      save:    MACRO ; Start macro definiti
    6    6                                 LDX    #storage
    7    7                                 LDAA   \1
    8    8                                 STAA   0,x ; Save first arg
    9    9                                 LDAA   \2
   10   10                                 STAA   2,x ; Save second ar
   11   11                                 IFC '\3', '' ; Is there a
   12   12                                   MEXIT    ; No, exit macro
   13   13                                 ENDC
   14   14                                 LDAA   \3    ; Save third ar
   15   15                                 STAA   4,X
   16   16                               ENDM          ; End of macro
   17   17
   18   18                      datSec:  SECTION
   19   19   000000             char1:   ds.b  1
   20   20   000001             char2:   ds.b  1
   21   21
   22   22                      codSec:  SECTION
   23   23                      entry:
   24   24                               save   char1, char2
   25    6m   000000 CE 00FF    +        LDX    #storage
   26    7m   000003 B6 xxxx    +        LDAA   char1
   27    8m   000006 6A00       +        STAA   0,x    ; save first arg
   28    9m   000008 B6 xxxx    +        LDAA   char2
   29   10m  00000B 6A02        +        STAA   2,x    ; save second ar
   30   11m         0000 0001   +        IFC '', ''    ; Is there a 3rd
   32   12m                     +          MEXIT       ; No, exit macro
   33   13m                     +        ENDC
   34   14m                     +        LDAA          ; Save third arg
   35   15m                     +        STAA   4,X
```

MLIST - List macro expansions

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`.

Syntax

```
MLIST [ON|OFF]
```

Synonym

None

Example

The assembly code in Listing 8.42, with `MLIST ON`, generates the assembler output listing in Listing 8.43.

Listing 8.42 Example assembly source code

```
        XDEF   entry
        MLIST  ON
swap:   MACRO
          LDD    \1
          LDX    \2
          STD    \2
          STX    \1
        ENDM
codSec: SECTION
entry:
        LDD   #$F0
        LDX   #$0F
main:
        STD   first
        STX   second
        swap  first, second
        NOP
        BRA   main
```

```
datSec: SECTION
first:  DS.W   1
second: DS.W   1
```

Listing 8.43 Assembler output listing the example in Listing 8.42 with MLIST ON

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                              XDEF  entry
    3    3                      swap:   MACRO
    4    4                                LDD   \1
    5    5                                LDX   \2
    6    6                                STD   \2
    7    7                                STX   \1
    8    8                              ENDM
    9    9                      codSec: SECTION
   10   10                      entry:
   11   11   000000 CC 00F0             LDD   #$F0
   12   12   000003 CE 000F             LDX   #$0F
   13   13                      main:
   14   14   000006 7C xxxx             STD   first
   15   15   000009 7E xxxx             STX   second
   16   16                              swap  first, second
   17    4m  00000C FC xxxx    +          LDD   first
   18    5m  00000F FE xxxx    +          LDX   second
   19    6m  000012 7C xxxx    +          STD   second
   20    7m  000015 7E xxxx    +          STX   first
   21   17   000018 A7                  NOP
   22   18   000019 20EB                BRA   main
   23   19                      datSec: SECTION
   24   20   000000             first:  DS.W  1
   25   21   000002             second: DS.W  1
```

Using the same code, with `MLIST OFF`, the Assembler produces the listing file shown in Listing 8.44.

Listing 8.44 Listing File with MLIST OFF

```
HC12-Assembler

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

```
  1    1                              XDEF   entry
  3    3                      swap:   MACRO
  4    4                                LDD    \1
  5    5                                LDX    \2
  6    6                                STD    \2
  7    7                                STX    \1
  8    8                              ENDM
  9    9                      codSec: SECTION
 10   10                      entry:
 11   11   000000 CC 00F0             LDD    #$F0
 12   12   000003 CE 000F             LDX    #$0F
 13   13                      main:
 14   14   000006 7C xxxx             STD    first
 15   15   000009 7E xxxx             STX    second
 16   16                              swap   first, second
 21   17   000018 A7                  NOP
 22   18   000019 20EB                BRA    main
 23   19                      datSec: SECTION
 24   20   000000             first:  DS.W   1
 25   21   000002             second: DS.W   1
```

The `MLIST` directive does not appear in the listing file. When a macro is called after a `MLIST ON`, it is expanded in the listing file. If the `MLIST OFF` is encountered before the macro call, the macro is not expanded in the listing file.

NOLIST - Disable listing

Description

Suppresses the printing of the following instructions in the assembly listing and debug file until a LIST - Enable listing assembler directive is reached.

Syntax

`NOLIST`

Synonym

`NOL`

Example

See Listing 8.45 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.

Listing 8.46 Assembler output listing from the assembler source code in Listing 8.45

```
HC12-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 A7          aaa:    NOP
    2    2
    4    4   000001 A7          bbb:    NOP
    5    5   000002 A7                  NOP
    6    6
   12   12   000005 A7          ddd:    NOP
   13   13   000006 A7                  NOP
```

See Also

LIST - Enable listing assembler directive

NOPAGE - Disable paging

Description

Disables pagination in the listing file. Program lines are listed continuously, without headings or top or bottom margins.

Syntax

```
NOPAGE
```

Synonym

None

OFFSET - Create absolute symbols

Description

The `OFFSET` directive declares an offset section and initializes the location counter to the value specified in `<expression>`. The `<expression>` must be absolute and may not contain references to external, undefined or forward defined labels.

An offset section is useful to simulate data structures or a stack frame.

Syntax

```
OFFSET <expression>
```

Synonym

None

Examples

The example shown in Listing 8.47 shows you how to use the `OFFSET` directive to access elements of a structure.

Listing 8.47 Using the OFFSET Directive

```
         OFFSET 0
ID:      DS.B   1
COUNT:   DS.W   1
```

```
VALUE:   DS.L  1
SIZE:    EQU   *

DataSec: SECTION
Struct:  DS.B  SIZE

CodeSec: SECTION
entry:
         LDX   #Struct
         LDAA  #0
         STAA  ID, X
         INC   COUNT, X
         INCA
         STAA  VALUE, X
```

When a statement affecting the location counter other than `EVEN`, `LONGEVEN`, `ALIGN`, or `DS` is encountered after the `OFFSET` directive, the offset section is terminated. The preceding section is reactivated, and the location counter is restored to the next available location in this section.

See Listing 8.48 for an example.

Listing 8.48 Example—Using the OFFSET Directive

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               OFFSET 0
    2    2   000000             ID:      DS.B   1
    3    3   000001             COUNT:   DS.W   1
    4    4   000003             VALUE:   DS.L   1
    5    5          0000 0007   SIZE:    EQU *
    6    6
    7    7                      DataSec: SECTION
    8    8   000000             Struct:  DS.B SIZE
    9    9
   10   10                      CodeSec: SECTION
   11   11                      entry:
   12   12   000000 CExx xx              LDX   #Struct
   13   13   000003 8600                 LDAA  #0
   14   14   000005 6A00                 STAA  ID, X
   15   15   000007 6201                 INC   COUNT, X
   16   16   000009 42                   INCA
   17   17   00000A 6A03                 STAA  VALUE, X
```

In the example above, the `cst3` symbol, defined after the `OFFSET` directive, defines a constant byte value. This symbol is appended to the `ConstSec` section, which precedes the `OFFSET` directive.

ORG - Set location counter

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).

Syntax

```
ORG <expression>
```

Synonym

None

Example

Listing 8.49 shows how to use ORG to set the location counter.

Listing 8.49 Using ORG to set the location counter

```
        org    $2000
b1:     nop
b2:     rts
```

Listing 8.50 shows 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

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----  ------ ---------   -----------
    1    1                            org   $2000
    2    2  a002000 A7          b1:   nop
    3    3  a002001 3D          b2:   rts
```

See also

- DC - Define Constant
- DCB - Define constant block
- DS - Define space
- SECTION - Declare relocatable section

PAGE - Insert page break

Description

Insert a page break in the assembly listing.

Syntax

```
PAGE
```

Synonym

None

Example

The portion of code in Listing 8.51 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.

Listing 8.52 Assembler output listing from the source code in Listing 8.51

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      code:    SECTION
    2    2   000000 0012                 DC.B   $00,$12
```

```
   3     3    000002 0034                  DC.B  $00,$34


 Abs. Rel.    Loc    Obj. code    Source line
 ---- ----    ------ ---------    -----------
    5     5   000004 0056                  DC.B    $00,$56
    6     6   000006 0078                  DC.B    $00,$78
```

PLEN - Set page length

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.

Syntax

```
PLEN <n>
```

Synonym

None

RAD50 - Rad50-encoded string constants

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).

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) about how to decode it.

Syntax

```
RAD50 <str>[, cnt]
```

Synonym

`None`

Example

The string data in Listing 8.54 assembles to the following data (Listing 8.55). 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 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

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.

Syntax

```
<name>: SECTION [SHORT][<number>]
```

Synonym

None

Examples

The example in Listing 8.57 demonstrates the definition of a section `aaa`, which is split in two blocks, with section `bbb` in between them.

The location counter associated with the label `zz` is 1, because a `NOP` instruction was already defined in this section at label `xx`.

Listing 8.57 Example of the SECTION assembler directive

```
    1    1                        aaa:    SECTION 4
    2    2   000000 A7            xx:     NOP
    3    3                        bbb:    SECTION 5
    4    4   000000 A7            yy:     NOP
    5    5   000001 A7                    NOP
    6    6   000002 A7                    NOP
    7    7                        aaa:    SECTION 4
    8    8   000001 A7            zz:     NOP
```

The optional qualifier `SHORT` specifies that the section is a short section, That means than the objects defined there can be accessed using the direct addressing mode.

Listing 8.58 demonstrates the definition and usage of a `SHORT` section. In this case, the symbol data is accessed using the direct addressing mode.

Listing 8.58 Using the direct addressing mode

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      dataSec: SECTION SHORT
```

```
   2    2   000000               data:     DS.B  1
   3    3
   4    4                        codeSec: SECTION
   5    5
   6    6                        entry:
   7    7   000000 87                      CLRA
   8    8   000001 5Axx                    STAA  data
```

See also

Assembler directives:

- ORG - Set location counter
- DC - Define Constant
- DCB - Define constant block
- DS - Define space

SET - Set symbol value

Description

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.

Syntax

```
<label>: SET <expression>
```

Synonym

None

Example

See Listing 8.59 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

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.

Syntax

```
SPC <count>
```

Synonym

None

TABS - Set tab length

Description

Sets the tab length to `<n>` spaces. The default tab length is eight. `<n>` may range from 0 to 128.

Syntax

```
TABS <n>
```

Synonym

None

TITLE - Provide listing title

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.

Syntax

```
TITLE <title>
```

Synonym

```
TTL
```

XDEF - External symbol definition

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.

Syntax

```
XDEF [.<size>] <label>[,<label>]...
```

where `<size> = B(direct), W (default)`, or `L`

Synonym

```
GLOBAL, PUBLIC
```

Example

See Listing 8.60 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

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.

Syntax

```
XREF [.<size>] <symbol>[,<symbol>]...
```

where `<size> = B(direct), W` (default), or `L`.

Synonym

EXTERNAL

Example

```
XREF OtherGlobal ; Reference "OtherGlobal" defined in
                 ; another module. (See the XDEF
                 ; directive example.)
```

XREFB - External reference for symbols located on the direct page

Description

This directive specifies symbols referenced in the current module but defined in another module. Symbols enumerated in an `XREFB` directive, can be accessed using the direct address mode. The list of symbols and corresponding 8-bit values is passed to the linker.

The number of symbols enumerated in an `XREFB` directive is only limited by the memory available at assembly time.

Syntax

```
XREFB <symbol>[,<symbol>]...
```

Synonym

None

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:

Listing 9.1 Example macro definition

```
MyMacro: MACRO
           DC.\0    \1, \2
         ENDM
```

When this macro is used in a program, e.g.,

```
MyMacro.B $10, $56
```

the Assembler expands it to:

```
DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator `\1` through `\9` and `\A` through `\Z`, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument itself is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

Macro Argument Grouping

To pass text including commas as a single macro argument, the Assembler supports a special syntax. This grouping starts with the [? prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, brackets, question marks and backward slashes can also be escaped with a backward slash as a prefix.

NOTE This escaping only takes place inside of `[? ?]` arguments. A backslash is only removed in this process if it is just before a bracket `([` or `])`, a question mark `(?)`, or a second backslash `(\)`.

Listing 9.2 Example macro definition

```
MyMacro:   MACRO
             DC     \1
           ENDM
MyMacro1:  MACRO
             \1
           ENDM
```

Listing 9.3 shows macro calls with more complicated arguments:

Listing 9.3 Macro calls for Listing 9.2

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro  [?$10, $56?]?]
```

```
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines (Listing 9.4):

Listing 9.4 Macro expansion of Listing 9.3

```
DC     $10, $56
DC     "[?"
DC     $10, $56
DC     $10, $56
```

For compatibility, the Macro Assembler also supports previous version's macro grouping with angle bracket syntax (Listing 9.5):

Listing 9.5 Angle bracket syntax

```
      MyMacro <$10, $56>
```

CAUTION However, this old syntax is ambiguous, as < and > are also used as compare operators. For example, Listing 9.6 does not produce the expected result.

Listing 9.6 Potential problem using the angle-bracket syntax

```
      MyMacro <1 > 2, 2 > 3> ; Wrong!
```

TIP Because of this, avoid the old angle brace syntax in new code. There is also an option to disable it explicitly.

See also the following assembler options:

- -CMacBrackets: Square brackets for macro arguments grouping
- -CMacAngBrack: Angle brackets for grouping Macro Arguments

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).

Listing 9.7 Clear macro definition

```
clear:   MACRO
           LDX    #\1
           LDAA   #16
\@LOOP:    CLR    1,X+
           DBNE   A,\@LOOP
         ENDM
```

This macro is called in the application (Listing 9.8).

Listing 9.8 Calling the clear macro

```
Data: Section
temporary: DS  16
data:      DS  16

Code: Section
         clear temporary
         clear data
```

The two macro calls of `clear` are expanded in the manner shown in Listing 9.9.

Listing 9.9 Example—Labels within Macros

```
HC12-Assembler
 Abs. Rel.   Loc    Obj. code    Source line
 ---- ----   ------ ---------    -----------
    1    1                       clear:   MACRO
```

```
   2    2                                  LDX   #\1
   3    3                                  LDAA  #16
   4    4                        \@LOOP:     CLR   1,X+
   5    5                                  DBNE  A,\@LOOP
   6    6                                ENDM
   7    7
   8    8                        Data: Section
   9    9   000000               temporary: DS    16
  10   10   000010               data:      DS    16
  11   11
  12   12                        Code: Section
  13   13                                clear temporary
  14    2m  000000 CE xxxx     +         LDX   #temporary
  15    3m  000003 8610        +         LDAA  #16
  16    4m  000005 6930        +_00001LOOP: CLR   1,X+
  17    5m  000007 0430 FB     +         DBNE  A,_00001LOOP
  18   14                                clear data
  19    2m  00000A CE xxxx     +         LDX   #data
  20    3m  00000D 8610        +         LDAA  #16
  21    4m  00000F 6930        +_00002LOOP: CLR   1,X+
  22    5m  000011 0430 FB     +            DBNE  A,_00002LOOP
```

Macro Expansion

When the Assembler reads a statement in a source program calling a previously defined macro, it processes the call as described in the following paragraphs.

The symbol table is searched for the macro name. If it is not in the symbol table, an undefined symbol error message is issued.

The rest of the line is scanned for arguments. Any argument in the macro call is saved as a literal or null value in one of the 35 possible parameter fields. When the number of arguments in the call is less than the number of parameters used in the macro the argument, which have not been defined at invocation time are initialize with `""` (empty string).

Starting with the line following the `MACRO` directive, each line of the macro body is saved and is associated with the named macro. Each line is retrieved in turn, with parameter designators replaced by argument strings or assembler-generated label strings.

Once the macro is expanded, the source lines are evaluated and object code is produced.

Nested Macros

Macro expansion is performed at invocation time, which is also the case for nested macros. If the macro definition contains nested macro call, the nested macro expansion takes place in line. Recursive macro call are also supported.

A macro call is limited to the length of one line, or 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: Not 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, e.g., HC12.
- The third line contains a copyright notice (Listing 10.1).

Listing 10.1 Example page header output

```
Demo Application
Freescale HC12-Assembler
(c) COPYRIGHT Freescale 1991-2005
```

Source Listing

The printed columns can be configured in various formats with the -Lasmc: Configure listing file assembler option. The default format of the source listing has five columns:

- Abs.,
- Rel.,
- Loc,
- Obj. Code, and
- Source Line.

Abs.

This column contains the absolute line number for each instruction. The absolute line number is the line number in the debug listing file, which contains all included files and where any macro calls have been expanded.

Listing 10.2 Example output listing - Abs. column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                     ;------------------------------
    2    2                     ; File: test.o
    3    3                     ;------------------------------
    4    4
    5    5                              XDEF  Start
    6    6                     MyData:  SECTION
    7    7   000000            char1:   DS.B  1
    8    8   000001            char2:   DS.B  1
    9    9                              INCLUDE "macro.inc"
```

```
  10    1i                    cpChar:  MACRO
  11    2i                               LDAA  \1
  12    3i                               STAA  \2
  13    4i                             ENDM
  14   10                     CodeSec: SECTION
  15   11                     Start:
  16   12                              cpChar  char1, char2
  17    2m  000000 B6 xxxx   +           LDAA  char1
  18    3m  000003 7A xxxx   +           STAA  char2
  19   13   000006 A7                  NOP
  20   14   000007 A7                  NOP
```

In the previous example, the line number displayed in the 'Abs.' column is incremented for each line.

Rel.

This column contains the relative line number for each instruction. The relative line number is the line number in the source file. For included files, the relative line number is the line number in the included file. For macro call expansion, the relative line number is the line number of the instruction in the macro definition. See Listing 10.3.

An `'i'` suffix is appended to the relative line number when the line comes from an included file. An `'m'` suffix is appended to the relative line number when the line is generated by a macro call.

Listing 10.3 Example listing file - Rel. column

```
 Abs. Rel.   Loc    Obj. code    Source line
 ---- ----   ------ ---------    -----------
    1    1                       ;------------------------------
    2    2                       ; File: test.o
    3    3                       ;------------------------------
    4    4
    5    5                                XDEF Start
    6    6                       MyData:  SECTION
    7    7   000000              char1:   DS.B   1
    8    8   000001              char2:   DS.B   1
    9    9                                INCLUDE "macro.inc"
   10   1i                       cpChar:  MACRO
   11   2i                                  LDAA \1
   12   3i                                  STAA \2
   13   4i                                ENDM
   14  10                        CodeSec: SECTION
   15  11                        Start:
   16  12                                 cpChar char1, char2
```

```
  17     2m   000000 B6 xxxx     +            LDAA char1
  18     3m   000003 7A xxxx     +            STAA char2
  19    13    000006 A7                     NOP
  20    14    000007 A7                     NOP
```

In the previous example, the line number displayed in the ‘`Rel.`’ column. represent the line number of the corresponding instruction in the source file.

‘`1i`’ on absolute line number 10 denotes that the instruction `cpChar: MACRO` is located in an included file.

‘`2m`’ on absolute line number 17 denotes that the instruction `LDAA  char1` is generated by a macro expansion.

Loc

This column contains the address of the instruction. For absolute sections, the address is preceded by an ‘`a`’ and contains the absolute address of the instruction. For relocatable sections, this address is the offset of the instruction from the beginning of the relocatable section. This offset is a hexadecimal number coded on 6 digits.

A value is written in this column in front of each instruction generating code or allocating storage. This column is empty in front of each instruction that does not generate code (for example `SECTION`, `XDEF`, …). See Listing 10.4.

Listing 10.4 Example Listing File - Loc column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ---------- ---------   -----------------------------
    1    1                          ;------------------------------
    2    2                          ; File: test.o
    3    3                          ;------------------------------
    4    4
    5    5                                   XDEF Start
    6    6                          MyData:  SECTION
    7    7   000000                 char1:   DS.B  1
    8    8   000001                 char2:   DS.B  1
    9    9                                   INCLUDE "macro.inc"
   10   1i                          cpChar:  MACRO
   11   2i                                     LDAA  \1
   12   3i                                     STAA  \2
   13   4i                                   ENDM
   14   10                          CodeSec: SECTION
   15   11                          Start:
   16   12                                   cpChar char1, char2
   17   2m   000000 B6 xxxx     +              LDAA  char1
   18   3m   000003 7A xxxx     +              STAA  char2
```

```
  19    13    000006 A7                    NOP
  20    14    000007 A7                    NOP
```

In the previous example, the hexadecimal number displayed in the column `Loc.` is the offset of each instruction in the section `codeSec`.

There is no location counter specified in front of the instruction `INCLUDE "macro.inc"` because this instruction does not generate code.

The instruction `LDAA  char1` is located at offset 0 from the section `codeSec` start address.

The instruction `STAA  char2` is located at offset 3 from the section `codeSec` start address.

Obj. Code

This column contains the hexadecimal code of each instruction in hexadecimal format. This code is not identical to the code stored in the object file. The letter ‘x’ is displayed at the position where the address of an external or relocatable label is expected. Code at any position when ‘x’ is written will be determined at link time. See Listing 10.5.

Listing 10.5 Example listing file - Obj. column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ----------- -----------
    1    1                      ;-------------------------------
    2    2                      ; File: test.o
    3    3                      ;-------------------------------
    4    4
    5    5                               XDEF Start
    6    6                      MyData:  SECTION
    7    7   000000             char1:   DS.B  1
    8    8   000001             char2:   DS.B  1
    9    9                               INCLUDE "macro.inc"
   10   1i                      cpChar:  MACRO
   11   2i                                 LDAA  \1
   12   3i                                 STAA  \2
   13   4i                               ENDM
   14   10                      CodeSec: SECTION
   15   11                      Start:
   16   12                               cpChar char1, char2
   17   2m   000000 B6 xxxx    +          LDAA char1
   18   3m   000003 7A xxxx    +          STAA char2
   19   13   000006 A7                   NOP
   20   14   000007 A7                   NOP
```

Source Line

This column contains the source statement. This is a copy of the source line from the source module. For lines resulting from a macro expansion, the source line is the expanded line, where parameter substitution has been done. See Listing 10.6.

Listing 10.6 Example listing file - Source line column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ;--------------------------------
    2    2                      ; File: test.o
    3    3                      ;--------------------------------
    4    4
    5    5                                XDEF Start
    6    6                      MyData:   SECTION
    7    7   000000             char1:    DS.B   1
    8    8   000001             char2:    DS.B   1
    9    9                                INCLUDE "macro.inc"
   10   1i                      cpChar:   MACRO
   11   2i                                  LDAA   \1
   12   3i                                  STAA   \2
   13   4i                                ENDM
   14   10                      CodeSec:  SECTION
   15   11                      Start:
   16   12                                cpChar char1, char2
   17   2m   000000 B6 xxxx    +            LDAA char1
   18   3m   000003 7A xxxx    +            STAA char2
   19   13   000006 A7                    NOP
   20   14   000007 A7                    NOP
```

11

Mixed C and Assembler Applications

When you intend to mix Assembly source file and ANSI-C source files in a single application, the following issues are important:

- Memory Models
- Parameter Passing Scheme
- Return Value
- Accessing Assembly Variables in an ANSI-C Source File
- Accessing ANSI-C Variables in an Assembly Source File
- Invoking an Assembly Function in an ANSI-C Source File
- Support for Structured Types

To build mixed C and Assembler applications, you have to know how the C Compiler uses registers and calls procedures. The following sections will describe this for compatibility with the compiler. If you are working with another vendor's ANSI-C compiler, refer to your Compiler Manual to get the information about parameter passing rules.

Memory Models

The memory models are only important if you mix C and assembly code. In this case all sources must be compiled or assembled with the same memory model.

The Assembler supports all memory models of the compiler. Depending on your hardware, use the smallest memory model suitable for your programming needs.

Table 11.1 summarizes the different memory models. It shows when to use a particular memory model and which assembler switch to use.

Table 11.1 S12(X) Memory Models

Option	Memory Model	Local Data	Global Data	Suggested Use
–Ms	SMALL	SP rel	extended	Small applications which fit into the 64k address space or which do only have limited places where paged area is accessed.
–Mb	BANKED	SP rel	extended	Larger applications which code does not fit into the 64k address space. Data is limited to the 64k address space. The code generated by the compiler is not much larger than in the SMALL memory model because the CPU supports the CALL instruction. Usually there is one additional byte per function call.
–Ml	LARGE	SP rel	far	Applications whose data does not fit into 64k address space. The code generated by the compiler is significantly larger than in the other memory models.

NOTE The default pointer size for the compiler is also affected by the memory model chosen.

Parameter Passing Scheme

When you are using the HC12 compiler, the parameter passing scheme is the following:

The Pascal calling convention is used for functions with a fixed number of parameters: The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack again.

The C calling convention is used only for functions with a variable number of parameters. In this case the caller pushes the arguments from right to left.

If the last parameter of a function with a fixed number of arguments has a simple type, it is not pushed but passed in a register. This results in shorter code because pushing the last parameter can be avoided. Table 11.2 shows an overview of the registers used for argument passing

Table 11.2 Registers Used for Passing the Last Argument to a Function

Size of Last Parameter	Type Example	Register
1 byte	char	B
2 bytes	int, array	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Parameters having a type not listed are passed on the stack (i.e., all those having a size greater than 4 bytes).

Return Value

Function results usually are returned in registers, except if the function returns a result larger than 4 bytes (see Table 11.3). Depending on the size of the return type, different registers are used:

Table 11.3 Data Type and Registers used in Function Returns

Size of Return Value	Type Example	Register
1 byte	char	B
2 bytes	int	D
3 bytes	far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Functions returning a result larger than two words are called with an additional parameter. This parameter is the address where the result should get copied to.

Accessing Assembly Variables in an ANSI-C Source File

A variable or constant defined in an assembly source file is accessible in an ANSI-C source file.

The variable or constant is defined in the assembly source file using the standard assembly syntax.

Variables and constants must be exported using the `XDEF` directive to make them visible from other modules (Listing 11.1).

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).

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).

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).

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).

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).

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

Listing 11.6 Example of data and constant reference

```
  LDAA CConst
....
  LDAA CData
 ....
```

Invoking an Assembly Function in an ANSI-C Source File

An function implemented in an assembly source file (`mixasm.asm` in Listing 11.7) can be invoked in a C source file (Listing 11.9). During the implementation of the function in the assembly source file, you should pay attention to the parameter passing scheme of the ANSI-C compiler you are using in order to retrieve the parameter from the right place.

Listing 11.7 Example of an assembly file: mixasm.asm

```
            XREF   CData
            XDEF   AddVar
            XDEF   ASMData

DataSec:    SECTION
ASMData:    DS.B   1
CodeSec:    SECTION
AddVar:
            ADDB   CData   ; add CData to the parameter in register B
            STAB   ASMData ; result of the addition in ASMData
            RTS
```

We recommend that you generate a header file for each assembly source file (Listing 11.7). This header file (`mixasm.h` in Listing 11.8) should contain the interface to the assembly module.

Listing 11.8 Header file for the assembly mixasm.asm file: mixasm.h

```
/* mixasm.h */
#ifndef _MIXASM_H_
#define _MIXASM_H_

void AddVar(unsigned char value);
/* function that adds the parameter value to global CData */
/* and then stores the result in ASMData */

/* variable which receives the result of AddVar */
extern char ASMData;

#endif /* _MIXASM_H_ */
```

The function can then be invoked in the usual way, by using its name.

Example of a C File

A C source code file (`mixc.c`) has the main() function which calls the `AddVar()` function. See Listing 11.9. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format).

Listing 11.9 Example C source code file: mixc.c

```
static int Error          = 0;
const unsigned char CData = 12;
#include "mixasm.h"

void main(void) {
  AddVar(10);
  if (ASMData != CData + 10){
    Error = 1;
  } else {
    Error = 0;
  }
  for(;;); // wait forever
}
```

NOTE Be careful, as the Assembler will not make any checks on the number and type of the function parameters.

The application must be correctly linked.

For these `C` and `*.asm` files, a possible linker parameter file is shown in Listing 11.10.

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.
- Define a structured variable. See Variable Definition.
- Declare a structured variable. See Variable Declaration.
- Access the address of a field inside of a structured variable. See Accessing a Field Address
- Access the offset of a field inside of a structured variable. See Accessing a Field Offset.

NOTE Some limitations apply in the usage of the structured types in the Macro Assembler. See Structured Type: Limitations.

Structured Type Definition

The Macro Assembler is extended with the following new keywords in order to support ANSI-C type definitions.

- `STRUCT`
- `UNION`

The structured type definition for `STRUCT` can be encoded as in Listing 11.11:

Listing 11.11 Definition for STRUCT

```
typeName: STRUCT
   lab1: DS.W 1
   lab2: DS.W 1
```

```
        . . .
ENDSTRUCT
```

where:

- `typeName` is the name associated with the defined type. The type name is considered to be a user-defined keyword. The Macro Assembler will be case-insensitive on `typeName`.
- `STRUCT` specifies that the type is a structured type.
- `lab1` and `lab2` are the fields defined inside of the `typeName type`. The fields will be considered as user-defined labels, and the Macro Assembler will be case-sensitive on label names.

As with all other directives in the Assembler, the `STRUCT` and `UNION` directives are case-insensitive.

The `STRUCT` and `UNION` directives cannot start on column 1 and must be preceded by a label.

Types Allowed for Structured Type Fields

The field inside of a structured type may be:

- another structured type or
- a base type, which can be mapped on 1, 2, or 4 bytes.

Table 11.4 shows how the ANSI-C standard types are converted in the assembler notation:

Table 11.4 Converting ANSI-C Standard Types to Assembler Notation

ANSI-C type	Assembler Notation
char	DS - Define space
short	DS.W
int	DS.W
long	DS.L
enum	DS.W
bitfield	-- not supported --
float	-- not supported --*DS.F*
double	-- not supported --*DS.D*

Table 11.4 Converting ANSI-C Standard Types to Assembler Notation (*continued*)

ANSI-C type	Assembler Notation
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):

`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 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), the Assembler uses the colon character ':'.

`var:field`

where

- '`var`' is the name of a variable, which was associated with a structured type.
- '`field`' is the name of a field in the structured type associated with the variable.

Listing 11.14 Example of accessing a field address

```
myType:  STRUCT
field1:    DS.W 1
field2:    DS.W 1
field3:    DS.B 1
field4:    DS.B 3
field5:    DS.W 1
         ENDSTRUCT

         XREF myData:myType
         XDEF entry

CodeSec: SECTION
entry:
         LDAA myData:field3 ; Loads register A with the
```

```
                              ; contents of field field3 from
                              ; variable myData.
```

NOTE The period cannot be used as separator because in assembly language it is a valid character inside of a symbol name.

Accessing a Field Offset

To access a structured type field offset, the Assembler will use following notation:

`<typeName>-><field>`

where:

- '`typeName`' is the name of a structured type.
- '`field`' is the name of a field in the structured type associated with the variable. See Listing 11.15 for an example of using this notation for accessing an offset.

Listing 11.15 Accessing a field offset with the -><field> notation

```
myType:   STRUCT
field1:     DS.W  1
field2:     DS.W  1
field3:     DS.B  1
field4:     DS.B  3
field5:     DS.W  1
          ENDSTRUCT
          XREF.B myData
          XDEF   entry

CodeSec:  SECTION
entry:
          LDX #myData
          LDAA myType->field3,X ; Adds the offset of field
                                ;'field3' (4) to X and loads
                                ; A with the content of the
                                ; effective address
```

Structured Type: Limitations

A field inside of a structured type may be:

- another structured type

- a base type, which can be mapped on 1, 2, or 4 bytes.

The Macro Assembler is not able to process bitfields or pointer types.

The type referenced in a variable definition or declaration must be defined previously. A variable cannot be associated with a type defined afterwards.

12

Make Applications

This chapters has the following sections:

Assembly Applications

This section covers:

Directly Generating an Absolute File

To use the Assembler to directly generate an absolute file, you must:

- Specify the application entry point in the assembly source file using the directive `ABSENTRY`.
- Encode the whole application in a single assembly unit.
- Ensure that the application contains only absolute sections.

To generate object files, the entry point of the application must be mentioned in the Linker parameter file using the `INIT` *funcname* command. The Linker builds the application using the different object files. See the Linker documentation for more information.

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. The Linker links all necessary object files — assembly or C — in the same way. See the Linker documentation for more information.

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).

Listing 12.1 SECTIONS/PLACEMENT portion of a PRM file

```
SECTIONS
  ROM1        = READ_ONLY  0x0200 TO 0x0FFF;
  SpecialROM = READ_ONLY  0x8000 TO 0x8FFF;
  RAM        = READ_WRITE 0x4000 TO 0x4FFF;
END

PLACEMENT
  DEFAULT_ROM   INTO ROM1;
  mySection     INTO SpecialROM;
  DEFAULT_RAM   INTO RAM;
END
```

13

How to...

This chapter covers the following topics:

- Working with Absolute Sections
- Working with Relocatable Sections
- Initializing the Vector Table
- Splitting an Application in Different Modules
- Using the Direct Addressing Mode to Access Symbols

Working with Absolute Sections

An absolute section is a section whose start address is known at assembly time. (See modules `fiboorg.asm` and `fiboorg.prm` in the demo directory.)

Defining Absolute Sections in an Assembly Source File

An absolute section is defined using the `ORG` directive. In that case, the Macro Assembler generates a pseudo section, whose name is `ORG_<index>`, where index is an integer which is incremented each time an absolute section is encountered (Listing 13.1).

Listing 13.1 Defining an absolute section containing data

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

In the previous portion of code, the `cst1` label is located at address `$A00`, and the `cst2` label is located at address `$A01`.

Listing 13.2 Assembler output listing for Listing 13.1

```
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).

Listing 13.3 Defining an absolute section containing code

```
        XDEF  entry
        ORG   $C00  ; Absolute code section.
entry:
LDAA    cst1        ; Load value in cst1
        ADDA  cst2  ; Add value in cst2
        STAA  var   ; Store in var
        BRA   entry
```

In the previous portion of code, the instruction LDAA will be located at address $C00, and instruction ADDA at address $C03. See Listing 13.4.

Listing 13.4 Assembler output listing for Listing 13.3

```
    6    6                                    ORG   $C00 ; Absolute section.
    7    7                             entry:
    8    8   a000C00 B6  0A00                 LDAA  cst1 ; Load value
    9    9   a000C03 BB  0A01                 ADDA  cst2 ; Add value in cst2
   10   10   a000C06 7A  0800                 STAA  var  ; Store in var
   11   11   a000C09 20F5                     BRA   entry
```

In order to avoid problems during linking or execution from an application, ensure that the assembly file at least:

- Initializes the stack pointer if the stack is used (use the instruction `LDS`).
- Publishes the application’s entry point using `XDEF`.

In addition, 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.

Listing 13.5 Minimal linker parameter file

```
LINK test.abs  /* Name of the executable file generated.      */
NAMES
  test.o       /* Name of the object files in the application. */
END

SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file.
*/
  MY_ROM  = READ_ONLY  0x4000 TO 0x4FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file.
*/
  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END

PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.          */
  DEFAULT_RAM    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM    INTO MY_ROM;
END
```

```
INIT entry                   /* Application entry point.          */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

CAUTION There should be no overlap between the absolute sections defined in the assembly source file and the memory areas defined in the PRM file.

NOTE As the memory areas (segments) specified in the PRM file are only used to allocate relocatable sections, nothing will be allocated there when the application contains only absolute sections. In that case you can even specify invalid address ranges in the PRM file.

Working with Relocatable Sections

A relocatable section is a section for which the start address is determined at linking time. (See modules `fibo.asm` and `fibo.prm` in the demo directory.)

Defining Relocatable Sections in a Source File

A relocatable section is defined using the `SECTION` directive. See Listing 13.6 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.

Listing 13.7 Assembler output listing for Listing 13.6

```
  2    2                       constSec: SECTION ; Relocatable
  3    3   000000 A6           cst1:     DC.B    $A6
  4    4   000001 BC           cst2:     DC.B    $BC
  5    5
  6    6                       dataSec:  SECTION ; Relocatable
  7    7   000000              var:      DS.B   1
```

Program assembly source code should be located in a separate relocatable section (Listing 13.8).

Listing 13.8 Defining a relocatable section for code

```
         XDEF entry
codeSec: SECTION     ; Relocatable code section.
entry:
         LDAA  cst1  ; Load value in cst1
         ADDA  cst2  ; Add value in cst2
         STAA  var   ; Store in var
         BRA   entry
```

In the previous portion of code, the instruction LDAA will be located at offset 0 from the section codeSec start address, and instruction ADDA at offset 3 from the section `codeSec` start address.

To avoid problems during linking or execution from an application, ensure that an assembly file at least:

- Initializes the stack pointer if the stack is used (use the instruction `LDS` to initialize the stack pointer).
- Publishes 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.

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 Ensure that the memory ranges specified in the `SECTIONS` block are valid addresses for the controller being used. In addition, when using the SDI debugger, ensure that the addresses specified for code or constant sections are located in the target board ROM area. Otherwise, the debugger will be unable to load the application.

The sample `main.asm` module created by the CodeWarrior New Project Wizard relocatable assembly project is an example of usage of relocatable sections in an application.

Initializing the Vector Table

The vector table can be initialized in the assembly source file or in the linker parameter file. We recommend that you initialize it in the linker parameter file.

- Initializing the Vector Table in the Linker PRM File (recommended),
- Initializing the Vector Table in a Source File using a Relocatable Section, or
- Initializing the Vector Table in a Source File using an Absolute Section.

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). All these labels must be published, otherwise they cannot be addressed in the linker PRM file.

Listing 13.10 Initializing the Vector table from a PRM File

```
         XDEF IRQFunc, XIRQFunc, SWIFunc, OpCodeFunc, ResetFunc
DataSec: SECTION
Data:    DS.W  5 ; Each interrupt increments an element in the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
         LDAB  #0
         BRA   int
XIRQFunc:
         LDAB  #2
         BRA   int
SWIFunc:
         LDAB  #4
         BRA   int
OpCodeFunc:
         LDAB  #6
         BRA   int
ResetFunc:
         LDAB  #8
         BRA   entry
int:
         LDX   #Data ; Load address of symbol Data in X
         ABX    ; X <- address of the appropriate element in the table
         INC   0, X  ; The table element is incremented
```

```
         RTI
entry:
         LDS #$AFE
loop:    BRA loop
```

NOTE The functions `IRQFunc`, `XIRQFunc`, `SWIFunc`, `OpCodeFunc`, `ResetFunc` are published. This is required because they are referenced in the linker PRM file.

NOTE The processor automatically pushes all registers onto the stack when an interrupt occurs. It is not necessary for the interrupt function to save and restore the registers being used.

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).

Listing 13.11 Using the VECTOR ADDRESS Linker Command

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0B00 TO 0x0CFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END

INIT ResetFunc
VECTOR ADDRESS 0xFFF2 IRQFunc
VECTOR ADDRESS 0xFFF4 XIRQFunc
VECTOR ADDRESS 0xFFF6 SWIFunc
VECTOR ADDRESS 0xFFF8 OpCodeFunc
VECTOR ADDRESS 0xFFFE ResetFunc
```

NOTE The statement `INIT ResetFunc` defines the application entry point. Usually, this entry point is initialized with the same address as the reset vector.

NOTE The statement `VECTOR ADDRESS 0xFFF2 IRQFunc` specifies that the address of the `IRQFunc` function should be written at address `0xFFF2`.

Initializing the Vector Table in a Source File using a Relocatable Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions that should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See Listing 13.12.

Listing 13.12 Initializing the Vector table in source code with a relocatable section

```
         XDEF  ResetFunc
DataSec: SECTION
Data:    DS.W  5  ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
         LDAB  #0
         BRA   int
XIRQFunc:
         LDAB  #2
         BRA   int
SWIFunc:
         LDAB  #4
         BRA   int
OpCodeFunc:
         LDAB  #6
         BRA   int
ResetFunc:
         LDAB  #8
         BRA   entry
DummyFunc:
         RTI
int:
```

```
         LDX   #Data
         ABX
         INC   0, X
         RTI
entry:
         LDS #$AFE
loop:    BRA loop

VectorTable:SECTION
; Definition of the vector table.
IRQInt:        DC.W  IRQFunc
XIRQInt:       DC.W  XIRQFunc
SWIInt:        DC.W  SWIFunc
OpCodeInt:     DC.W  OpCodeFunc
COPResetInt:   DC.W  DummyFunc  ; No function attached to COP Reset.
ClMonResInt:   DC.W  DummyFunc  ; No function attached to Clock
                                ; MonitorReset.
ResetInt:      DC.W  ResetFunc
```

NOTE Each constant in the `VectorTable` section is defined as a word (a 2-byte constant), because the entries in the vector table are 16 bits wide.

NOTE In the previous example, the constant `IRQ1Int` is initialized with the address of the label `IRQ1Func`.

NOTE In the previous example, the constant `XIRQInt` is initialized with the address of the label `XIRQFunc`.

NOTE All the labels specifying an initialization value must be defined, published (using `XDEF`), or imported (using `XREF`) in the assembly source file.

Now place the section at the expected address using the linker parameter file (Listing 13.13).

Listing 13.13 Example linker parameter file

```
LINK  test.abs
NAMES test.o
END
SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
```

```
   /* Define the memory range for the vector table */
  Vector   = READ_ONLY  0xFFF2 TO 0xFFFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
   /* Place the section 'VectorTable' at the appropriated address. */
  VectorTable        INTO Vector;
END

INIT ResetFunc
ENTRIES
  *
END
```

NOTE The statement `Vector = READ_ONLY 0xFFF2 TO 0xFFFF` defines the memory range for the vector table.

NOTE The statement `VectorTable INTO Vector` specifies that the `VectorTable` section should be loaded in the read only memory area `Vector`. This means the constant `IRQInt` will be allocated at address `0xFFF2`, the constant `XIRQInt` will be allocated at address `0xFFF4`, and so on. The constant `ResetInt` will be allocated at address `0xFFFE`.

NOTE The statement `ENTRIES * END` switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

NOTE When developing a banked application, make sure that the code from the interrupt functions is located in the non banked memory area.

Initializing the Vector Table in a Source File using an Absolute Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables. See Listing 13.14 for an example.

Listing 13.14 Initializing the Vector table using an absolute section

```
         XDEF  ResetFunc
DataSec: SECTION
Data:    DS.W  5 ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
           LDAB  #0
           BRA   int
XIRQFunc:
           LDAB  #2
           BRA   int
SWIFunc:
           LDAB  #4
           BRA   int
OpCodeFunc:
           LDAB  #6
           BRA   int
ResetFunc:
           LDAB  #8
           BRA   entry

DummyFunc:
           RTI
int:
           LDX   #Data
           ABX
           INC   0, X
           RTI
entry:
           LDS  #$AFE
loop:      BRA   loop

           ORG   $FFF2
;Definition of the vector table
;in an absolute section starting at address $FFF2.
IRQInt:        DC.W IRQFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
OpCodeInt:     DC.W OpCodeFunc
COPResetInt:   DC.W DummyFunc  ; No function attached to COP Reset.
ClMonResInt:   DC.W DummyFunc  ; No function attached to Clock
```

```
                                  ; MonitorReset.
ResetInt:      DC.W ResetFunc
```

Now place the section at the expected address using the linker parameter file (Listing 13.15).

NOTE Each constant in the section starting at `$FFF2` is defined as a word (a 2-byte constant), because the entry in the vector table are 16 bits wide.

NOTE In the previous example, the constant `IRQInt` is initialized with the address of the label `IRQFunc`.

NOTE All the labels with an initialization value must be defined, published (using `XDEF`) or imported (using `XREF`) in the assembly source file.

NOTE The statement `ORG  $FFF2` specifies that the following section must start at address `$FFF2`.

Listing 13.15 Example linker parameter file for Listing 13.14

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0A00 TO 0x0BFF;
END
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END

INIT ResetFunc
ENTRIES
  *
END
```

NOTE The statement `ENTRY * END` switches smart linking off. If this statement is missing in the PRM file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

NOTE When developing a banked application, make sure that the code from the interrupt functions is located in the non-banked memory area

Splitting an Application in Different Modules

Complex applications or applications involving several programmers can be split into several simple modules. To avoid any problems when merging the different modules follow these rules:

- 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)

Listing 13.16 is an example assembly file which is used in the following sections.

Listing 13.16 Separating Code into Modules—Test1.asm

```
           XDEF AddSource
           XDEF Source

initStack: EQU $AFF

DataSec:   SECTION
Source:    DS.B 1
CodeSec:   SECTION
AddSource:
           ADDA Source
           STAA Source
           RTS
```

Corresponding Include File (Test1.inc)

See Listing 13.17 for an example Test1inc include file.

Listing 13.17 Separating Code into Modules—Test1.inc

```
         XREF AddSource
; The AddSource function adds the value stored in the variable
; Source to the contents of the A register. The result of the
; computation is stored in the Source variable.
;
; Input Parameter:  The A register contains the value that should be
;                   added to the Source variable.
; Output Parameter: Source contains the result of the addition.

         XREF Source
; The Source variable is a 1-byte variable.
```

Example of an Assembly File (Test2.asm)

Listing 13.18 is another assembly code file module for this project.

Listing 13.18 Separating Code into Modules—Test2.asm

```
           XDEF entry
           INCLUDE "Test1.inc"

initStack: EQU $AFE

CodeSec:   SECTION
entry:     LDS #initStack
           LDAA #$7
           JSR AddSource
           BRA entry
```

The application's `*.prm` file should list both object files building the application. When a section is present in the different object files, the object file sections are concatenated into a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the `*.prm` file.

Example of a PRM File (`Test2.prm`)

Listing 13.19 Separating assembly code into modules—Test2.prm

```
LINK test2.abs /* Name of the executable file generated. */
NAMES
  test1.o
  test2.o / *Name of the object files building the application. */
END

SECTIONS
  MY_ROM   = READ_ONLY  0x2B00 TO 0x2BFF; /* READ_ONLY  mem. */
  MY_RAM   = READ_WRITE 0x2800 TO 0x28FF; /* READ_WRITE mem. */
END

PLACEMENT
  /* variables are allocated in MY_RAM          */
  DataSec, DEFAULT_RAM           INTO MY_RAM;
  /* code and constants are allocated in MY_ROM */
  CodeSec, ConstSec, DEFAULT_ROM INTO MY_ROM;
END
INIT  entry      /* Definition of the application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Definition of the reset vector. */
```

NOTE The `CodeSec` section is defined in both object files. In `test1.o`, the `CodeSec` section contains the symbol `AddSource`. In `test2.o`, the `CodeSec` section contains the `entry` symbol. According to the order in which the object files are listed in the `NAMES` block, the function `AddSource` is allocated first and the `entry` symbol is allocated next to it.

Using the Direct Addressing Mode to Access Symbols

There are different ways for the Assembler to use the direct addressing mode on a symbol:

- Using the Direct Addressing Mode to Access External Symbols,
- Using the Direct Addressing Mode to Access Exported Symbols,
- Defining Symbols in the Direct Page,
- Using the Force Operator, or
- Using SHORT Sections.

Using the Direct Addressing Mode to Access External Symbols

External symbols, which should be accessed using the direct addressing mode, must be declared using the `XREF.B` directive. Symbols which are imported using `XREF` are accessed using the extended addressing mode.

Listing 13.20 Using direct addressing to access external symbols

```
     XREF.B ExternalDirLabel
     XREF   ExternalExtLabel

     LDD    ExternalDirLabel ; Direct addressing mode is used.

     LDD    ExternalExtLabel ; Extended addressing mode is used.
```

Using the Direct Addressing Mode to Access Exported Symbols

Symbols, which are exported using the `XDEF.B` directive, will be accessed using the direct addressing mode. Symbols which are exported using `XDEF` are accessed using the extended addressing mode.

Listing 13.21 Using direct addressing to access exported symbols

```
XDEF.B DirLabel
XDEF   ExtLabel
…
LDD   DirLabel ; Direct addressing mode is used.
…
LDD   ExtLabel ; Extended addressing mode is used.
```

Defining Symbols in the Direct Page

Symbols that are defined in the predefined `BSCT` section are always accessed using the direct-addressing mode (Listing 13.22).

Listing 13.22 Defining symbols in the direct page

```
…
          BSCT
```

```
DirLabel: DS.B   3
dataSec:  SECTION
ExtLabel: DS.B   5
…
codeSec:  SECTION
…
          LDD   DirLabel ; Direct addressing mode is used.
…
          LDD   ExtLabel ; Extended addressing mode is used.
```

Using the Force Operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode (Listing 13.23).

The supported force operators are:

- < or `.B` to force direct addressing mode
- > or `.W` to force extended addressing mode.

Listing 13.23 Using a force operator

```
…
dataSec: SECTION
label:   DS.B   5
…
codeSec: SECTION
…
         LDD   <label   ; Direct addressing mode is used.
         LDD   label.B ; Direct addressing mode is used.
…
         LDD   >label   ; Extended addressing mode is used.
         LDD   label.W ; Extended addressing mode is used.
```

Using SHORT Sections

Symbols that are defined in a section which has the `SHORT` qualifier are always accessed using the direct addressing mode (Listing 13.24).

Listing 13.24 Using SHORT sections

```
…
shortSec: SECTION SHORT
```

```
DirLabel: DS.B  3
dataSec:  SECTION
ExtLabel: DS.B  5
…
codeSec:  SECTION
…
          LDD   DirLabel ; Direct addressing mode is used.
…
          LDD   ExtLabel ; Extended addressing mode is used.
```

II

Appendices

This document has the following appendices:

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
- [Options] Section
- [XXX_Assembler] Section
- [Editor] Section

[Installation] Section

Path

Description

Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

Arguments

Last installation path.

Example

```
Path=C:\install
```

Group

Description

Whenever a tool is installed, the installation script stores the installation program group created into this variable.

Arguments

Last installation program group.

Example

```
Group=Assembler
```

[Options] Section

DefaultDir

Description

Specifies the current directory for all tools on a global level. See also DEFAULTDIR: Default current directory environment variable.

Arguments

Default Directory to be used.

Example

```
DefaultDir=C:\install\project
```

[XXX_Assembler] Section

This section documents the entries that can appear in an `[XXX_Assembler]` section of the `mcutools.ini` file.

NOTE *XXX* is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC12 Assembler, the name of this section would be [HC12_Assembler].

SaveOnExit

Description

Stores the configuration when the Assembler is closed.

Arguments

1/0

Remarks

1 to store the configuration when the Assembler is closed, 0 to discard the configuration. The Assembler does not ask to store a configuration in either case.

SaveAppearance

Description

Saves the appearance of the project file.

Arguments

1/0

Remarks

1 to store the visible topics when writing a project file, 0 to discard. The command line, its history, the windows position and other topics belong to this entry.

This entry corresponds to the state of the *Appearance* checkbox in the *Save Configuration* dialog box.

SaveEditor

Description

Saves the project file editor settings.

Arguments

1/0

Remarks

1 to store the editor settings when writing a project file, 0 to discard. The editor settings contain all information in the editor configuration dialog box. This entry corresponds to the state of the *Editor Configuration* checkbox in the *Save Configuration* dialog box.

SaveOptions

Description

Saves the project file options.

Arguments

1/0

Remarks

1 to save the options when writing a project file, 0 to discard. This entry corresponds to the state of the *Options* checkbox in the *Save Configuration* dialog box.

RecentProject0, RecentProject1

Description

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Arguments

Names of the last and prior project files

Example

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

[Editor] Section

Editor_Name

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.

Arguments

The name of the global editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Editor_Exe

Description

Specifies the filename which is started to edit a text file, when the global editor setting is active.

Arguments

The name of the executable file of the global editor (including path).

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Editor_Opts

Description

Specifies options (arguments), to use 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.

Arguments

The options to use with the global editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Example

```
[Editor]
editor_name=WinEdit
editor_exe=C:\WinEdit32\WinEdit.exe
editor_opts=%f /#:%l
```

Example

Listing A.1 shows a typical `mcutools.ini` file.

Listing A.1 Typical mcutools.ini file layout

```
[Installation]
Path=c:\Freescale
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\WinEdit32\WinEdit.exe
editor_opts=%f /#:%l

[Options]
DefaultDir=c:\myprj

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

B

Local Configuration File Entries

This appendix documents the sections and entries that can appear in the local configuration file. Usually, you name this file `project.ini`, where *`project`* is a placeholder for the name of your project.

A *`project`*`.ini` file could contain these sections for using the Assembler:

- [Editor] Section
- [XXX_Assembler] Section

See the Example section for a sample `project.ini` file.

[Editor] Section

Editor_Name

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.

Arguments

The name of the local editor

Saved

Only with `'Editor Configuration'` set in the *File > Configuration Save Configuration* dialog box.

Editor_Exe

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.

Arguments

The name of the executable file of the local editor (including path).

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Editor_Opts

Description

Specifies options (arguments), which should be used when starting the local editor. If this entry is absent or is 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.

Arguments

The options to use with the local editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Example

```
[Editor]
editor_name=WINEdit
editor_exe=C:\WinEdit32\WinEdit.exe
editor_opts=%f /#:%l
```

[XXX_Assembler] Section

This section documents the entries that can appear in an `[XXX_Assembler]` section of a `project.ini` file.

NOTE *XXX* is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC12 Assembler, the name of this section would be `[HC12_Assembler]`.

RecentCommandLineX, X= integer

Arguments

String with a command line history entry, e.g., `fibo.asm`

Description

This list of entries contains the content of the command line history.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

CurrentCommandLine

Description

The currently visible command line content.

Arguments

String with the command line, e.g., "`fibo.asm -w1`"

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

StatusbarEnabled

Description

Current status bar state.

- 1: Status bar is visible
- 0: Status bar is hidden

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it any more.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

ToolbarEnabled

Description

Current toolbar state

- 1: Toolbar is visible
- 0: Toolbar is hidden

Arguments

1/0

Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

WindowPos

Description

This contains the position and the state of the window (maximized, etc.) and other flags.

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.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

WindowFont

Description

Font attributes.

Arguments

size: == 0 -> generic size, < 0 -> font character height, > 0 -> font cell height

weight: 400 = normal, 700 = bold (valid values are 0 through 1000)

italic: 0 == no, 1 == yes

font name: max. 32 characters.

Saved

Only with *Appearance* set in the *File > Configuration Save Configuration* dialog box.

Example

```
WindowFont=-16,500,0,Courier
```

TipFilePos

Description

Actual position in tip of the day file. Used so that different tips are shown at different calls.

Arguments

any integer, e.g., 236

Saved

Always when saving a configuration file.

ShowTipOfDay

Description

Should the Tip of the Day dialog box be shown at startup?

- 1: Shown at startup
- 0: Show only when opened in the help menu

Arguments

0/1

Saved

Always when saving a configuration file.

Options

Description

The currently active option string. This entry can be very long.

Arguments

current option string, e.g.: `-W2`

Saved

Only with *Options* set in the *File > Configuration Save Configuration* dialog box.

EditorType

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 Settings Dialog Box.

Arguments

0/1/2/3/4

Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

EditorCommandLine

Description

Command line content to open a file. For details, see also Editor Settings Dialog Box.

Arguments

command line, for `WinEdit: "C:\WinEdit32\WinEdit.exe %f /#:%l"`

Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

EditorDDEClientName

Description

Name of the client for DDE editor configuration. For details, see also Editor Settings Dialog Box.

Arguments

client command, e.g., `"[open(%f)]"`

Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

EditorDDETopicName

Description

Name of the topic for DDE editor configuration. For details, see also Editor Settings Dialog Box.

Arguments

topic name, e.g., "system"

Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

EditorDDEServiceName

Description

Name of the service for DDE editor configuration. For details, see also Editor Settings Dialog Box.

Arguments

service name, e.g., `system`

Saved

Only with *Editor Configuration* set in the *File > Configuration Save Configuration* dialog box.

Example

The example in Listing B.1 shows a typical layout of the configuration file (usually `project.ini`).

Listing B.1 Example of a project.ini file

```
[Editor]
Editor_Name=IDF
Editor_Exe=C:\WinEdit32\WinEdit.exe
Editor_Opts=%f /#:%l

[XXX_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.asm -w2
RecentCommandLine1=fibo.asm
CurrentCommandLine=fibo.asm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\WinEdit32\WinEdit.exe %f /#:%l
```

C

MASM Compatibility

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

Comment Line

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

Constants (Integers)

For compatibility with the MASM Assembler, the following notations are also supported for integer constants (Listing C.1):

- A decimal constant is defined by a sequence of decimal digits (`0-9`) followed by a `d` or `D` character.
- A hexadecimal constant is defined by a sequence of hexadecimal digits (`0-9`, `a-f`, `A-F`) followed by a `h` or `H` character.
- An octal constant is defined by a sequence of octal digits (`0-7`) followed by an `o`, `O`, `q`, or `Q` character.
- A binary constant is defined by a sequence of binary digits (`0-1`) followed by a `b` or `B` character.

Listing C.1 Integer examples

```
512d        ; decimal representation
512D        ; decimal representation
200h        ; hexadecimal representation
200H        ; hexadecimal representation
1000o       ; octal representation
1000O       ; octal representation
1000q       ; octal representation
1000Q       ; octal representation
1000000000b ; binary representation
1000000000B ; binary representation
```

Operators

For compatibility with the MASM Assembler, the notations in Table C.1 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 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.
GLOBAL	XDEF	Make a symbol public (Visible from outside)

Table C.2 Supported MASM directives (*continued*)

Operator	Notation	Description
PUBLIC	XDEF	Make a symbol public (Visible from outside)
EXTERNAL	XREF	Import reference to an external symbol.
XREFB	XREF.B	Import reference to an external symbol located on the direct page.
SWITCH		Allows 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 covers the following topics:

- Labels
- SET Directive
- Obsolete Directives

Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even when they start on column 1.

When MCUasm compatibility mode is activated, following portion of code generate an error message, because the label `label` is not followed by a colon (Listing D.1).

Listing D.1 Erroneous label for MCUasm compatibility

```
label      DC.B   1
```

When MCUasm compatibility mode is not activated, the previous portion of code does not generate any error message.

SET Directive

When MCUasm compatibility mode is activated, relocatable expressions are also allowed in a SET directive.

When MCUasm compatibility mode is activated, the following portion of code does not generate any error messages (Listing D.2):

Listing D.2 SET directive

```
label: SET *
```

When MCUasm compatibility mode is not activated, the previous portion of code generates an error message because the `SET` label can only refer to absolute expressions.

Obsolete Directives

Table D.1 enumerates the directives, which are not recognized any longer when the MCUasm compatibility mode is switched ON.:

Table D.1 Obsolete Directives

Operator	Notation	Description
RMB	DS	Define storage for a variable
NOL	NOLIST	Specify that all subsequent instructions must not be inserted in the listing file.
TTL	TITLE	Define the user-defined title for the assembler listing file.
GLOBAL	XDEF	Make a symbol public (Visible from the outside)
PUBLIC	XDEF	Make a symbol public (Visible from the outside)
EXTERNAL	XREF	Import reference to an external symbol.

E

Semi-Avocet Compatibility

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

Avocet compatibility mode can be activated, specifying the -C=SAvocet: Switch Semi-Compatibility with Avocet Assembler ON assembler option. The compatibility does not cover all specific Avocet features but only some of them.

- Directives
- Section Definition
- Macro Parameters
- Support for Structured Assembly.

Directives

Table E.1 enumerates the directives which are supported when the Avocet Assembler compatibility mode is activated.

Table E.1 Avocet Assembler Directives

Directive	Notation	Description
DEFSEG		Segment definition (see Section Definition).

ELSEIF		Conditional directive, checking a specific condition.
		`IF ((label1 & label2) != 0)`
		`  LDD   #label1`
		`ELSIF (label1 = 0)`
		`  LDD   #label2`
		`ELSE`
		`  LDD   #0`
		`ENDIF`

Table E.1 Avocet Assembler Directives (*continued*)

Directive	Notation	Description

EXITM	MEXIT	Define an exit condition for a macro.
		`Copy      MACRO source, dest`
		`            IFB "source"`
		`              EXITM`
		`            ENDIF`
		`            LDD   source`
		`            STD   dest`
		`          ENDM`

IFB Param	IFC Param, ""	Test if a macro parameter is empty. The syntax is IFB "param".
		`Copy      MACRO   source, des`
		`            IFB   "source"`
		`              LDD   #0`
		`              STD   dest`
		`            ELSE`
		`              LDD   source`
		`              STD   dest`
		`            ENDIF`
		`          ENDM`

IFNB Param	IFNC Param ""	Test if a macro parameter is not empty. The syntax is IFNB "param"
		`Copy        MACRO   source, dest`
		`              IFNB  "source"`
		`              LDD   source`
		`              STD   dest`
		`            ELSE`
		`              LDD   #0`
		`              STD   dest`
		`            ENDIF`
		`          ENDM`

Table E.1 Avocet Assembler Directives (*continued*)

Directive	Notation	Description
NOSM	MLIST OFF	Do not insert the macro expansion in the listing file.
SEG	SWITCH	Switch to a previously defined segment. See Section Definition.
SM	MLIST ON	Insert the macro expansion in the listing file.
SUBTITLE		Defines a subtitle for the input file. This subtitle is written to the listing file. `SUBTITLE title2: Main File`
TEQ	SET	Define a constant, which value may be modified in the source file.

Section Definition

Section definition is performed using the `DEFSEG` directive. The correct syntax for a `DEFSEG` directive is:

```
DEFSEG <name> [START=<start address>] [<section qualifier>]
```

where:

- `name`: is the name of the section
- `start address`: is the start address for the section. This parameter is optional.
- `section qualifier`: is the qualifier which applies to the section. This parameter is optional and may take the value:

Table E.2 Section qualifiers

Qualifier	Meaning
PAGE0	for a data section located on the direct page
DATA	for a data section
CODE	for a code section

Some examples of the DEFSEG directive are shown in Listing E.1.

Listing E.1 DEFSEG examples

```
  DEFSEG myDataSection
  DEFSEG D_ATC_TABLES START=$0EA0
  DEFSEG myDirectData PAGE0
```

NOTE Because of an incompatibility in the object file format, an absolute section implementation must reside entirely in a single assembly unit. You cannot split the code from an absolute section over several object files. An absolute section is a section associated with a `start address`.

NOTE In order to split a section over different assembly units, you should define the section as relocatable (without `START`) and specify the address where you want to load the section in the linker PRM file.

The assembly source code in Listing E.2 relates to a possible allocation of memory as shown in Listing E.3.

Listing E.2 Example assembly code

```
DEFSEG D_ATC_TABLES ; START=$0EA0
```

Listing E.3 Portion of a linker parameter file

```
...
SECTION
  ...
  MY_TABLE = READ_WRITE 0x0EA0 TO 0x0EFF;
PLACEMENT
...
D_ATC_TABLES INTO MY_TABLE:
...
```

The `SEG` directive is then used to activate the corresponding section in the assembly source file.

The name specified in a `SEG` directive was previously specified in a `DEFSEG` directive.

The following syntax is acceptable for using the `SEG directive`:

```
SEG <name>
```

where:

`name:` is the name of the section, which was previously defined in a `DEFSEG` directive (Listing E.4).

Listing E.4 Example of using the SEG directive

```
SEG myDataSection
```

Macro Parameters

When Avocet Compatibility is switched `ON`, names can be associated with macro parameters. A macro definition could be as in Listing E.5:

Listing E.5 Example macro definition

```
Copy        MACRO    source, destination
              LDD    source
              STD    destination
            ENDM
```

Support for Structured Assembly

When the Avocet compatibility is switched on, the `SWITCH` or `FOR` constructs are available in Macro Assembler.

SWITCH Block

The `SWITCH` directive evaluates an expression and assembles the code following the particular `CASE` statement which satisfies the switch expression. If no `CASE` statement corresponds to the value of the expression, the code following the `DEFAULT` (if present) is assembled.

`ENDSW` terminates the `SWITCH` directive.

The expression specified in a `SWITCH` directive must be an absolute expression (Listing E.6).

Listing E.6 Example of using a SWITCH block

```
xxx    equu    5
  …
  SWITCH xxx
    CASE 0
      LDD    #1
    CASE 1
      LDD    2
    CASE 3
      LDD    #6
    DEFAULT
      LDD #0
  ENDSW
```

The instructions in Listing E.7 are generated by the code in Listing E.6. Assuming that the value for `xxx` was still 5 when the `SWITCH` statement was encountered, there was no particular result for `xxx` equal to 5, so the result for the `DEFAULT CASE` ensues:

`- LDD    #0.`

Listing E.7 Result of the SWITCH statement when xxx = 5

```
xxx    equ    5
  …
  LDD    #0
```

FOR Block

In the Avocet compatibility mode, the FOR - Repeat assembly block assembler directive is supported (Listing E.8).

Listing E.8 Example

```
  FOR l=2 TO 6
    NOP
  ENDFOR
```

The code segment in Listing E.8 generates the instructions in Listing E.9.

Listing E.9

```
    NOP
    NOP
    NOP
    NOP
    NOP
```

F

Using the Linux Command Line Assembler

The Linux version of the S12(X) assembler command line program is named `ahc12` and is located in the `prog` subfolder of the CodeWarrior installation path. The assembler program can be ran from a shell command line or specified in a makefile.

Command Line Arguments

Enter `ahc12 -h` to display a list of available arguments and options. Assembler options are documented in Assembler Options.

Command Examples

The following examples demonstrate some simple uses of the linux version of the HC12 command line assembler. Not all options or variations of options are provided.

Example of Setting CPU Option

The -Cpu option controls whether code for an HC12, an HCS12, or for an HCS12X is produced. To produce code for the HCS12X processor enter:

```
ahc12 main.asm -CpuHCS12X
```

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

Setting Maximum Number of Error Messages and Creating Error File

To set the maximum number of error messages to 5 and create the `err.log` error file and a listing file in the current directory enter:

```
ahc12 main.asm -WmsgNe5 -WErrFileOn -L
```

Displaying Environment Settings and Version Information

To display current environment variable settings such as LIBPATH and OBJPATH and version information enter:

```
ahc12 -v
```

Setting Color of Error Messages

The color setting options such as `-WmsgCE` are available for the Windows operating system only.

Using a Makefile

The GNU `make` command allows you to control and define the build process. The make program reads a file called makefile or Makefile. This file determines the relationships between the source, object and executable files.

Once you have created your Makefile and your corresponding source files, you are ready to use the GNU `make` command. If you have named your Makefile either Makefile or makefile, `make` will recognize it. If `make` does not recognize your makefile or it uses a different name, you can specify `make -f mymakefile`. The order in which dependencies are listed is important. If you simply type `make` and then return, `make` will attempt to create or update the first dependency listed.

Index

D

E

F

G

H

I

L

M