Freescale Semiconductor, Inc...

MCUez

Easy development software

from the company that

knows MCU hardware best

MCUez HC12 Assembler User's Manual
MCUEZASM12/D
Rev. 1

MOTOROLA

MCUez
HC12 Assembler

User's Manual

MOTOROLA

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

The computer program contains material copyrighted by Motorola, Inc., first published in 1997, and may be used only under a license such as the License For Computer Programs (Article 14) contained in Motorola's Terms and Conditions of Sale, Rev. 1/79.

Trademarks

This document includes these trademarks:

MCUez and MCUasm are trademarks of Motorola, Inc.

Microsoft Windows and Microsoft Developer Studio are registered trademarks of Microsoft Corporation in the U.S. and other countries.

WinEdit is a trademark of Wilson WindowWare.

For More Information On This Product,
Go to: www.freescale.com

List of Sections

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table of Contents

Section 1. General Information

Section 2. Graphical User Interface

Section 3. Environment Variables

Section 4. Files

Section 5. Assembler Options

Section 6. Sections

Section 7. Assembler Syntax

Section 8. Assembler Directives

Section 9. Macros

Freescale Semiconductor, Inc.

Section 10. Assembler Listing File

Section 11. Operating Procedures

Section 12. Assembler Messages

Freescale Semiconductor, Inc.

Appendix A. MASM Compatibility

Appendix B. MCUasm Compatibility

Index

List of Figures

List of Figures

List of Tables

Freescale Semiconductor, Inc.

Section 1. General Information

1.1 Contents

1.2 Introduction

Features of the MCUez HC12 assembler include:

- Graphical user interface (GUI)

- Online help

- Support for absolute and relocatable assembler code

- 32-bit application

- Compatible with MCUasm™ Release 5.3

- Conforms to Motorola assembly language input standard and *ELF/DWARF 2.0* object code format

1.3 Structure of This Manual

This list describes the topics contained in this manual.

- **Graphical user interface** — Description of the MCUez assembler GUI

- **Environment** — Description of the MCUez assembler environment variables

- **Assembler options** — Detailed description of the full set of assembler options

- **Assembler syntax** — Description of the assembler input file syntax

- **Assembler directives** — List of all directives supported by the assembler

- **Assembler messages** — Description and examples produced by the assembler

- **Appendices**

- **Index**

1.4 Getting Started

This section describes how to get started using MCUez. The locations of specific working directories and the directories reflected in dialog window reflect the directories that have been chosen.

This section provides instructions to:

- Create a new project

- Write the assembly source file

- Assemble the assembly source file

- Link the application to generate an executable file

NOTE: *All directory paths and listings are examples only. Paths and directory listings may change depending upon the MCUez configuration.*

1.4.1 Creating a New Project

The first step in creating an application is to define the new project. Do this by using the **MCUez Shell**.

1. Start the **MCUez Shell**.

Figure 1-1. MCUez Shell

2. Click on the **ezMCU** button to open the **Configuration** dialog box.
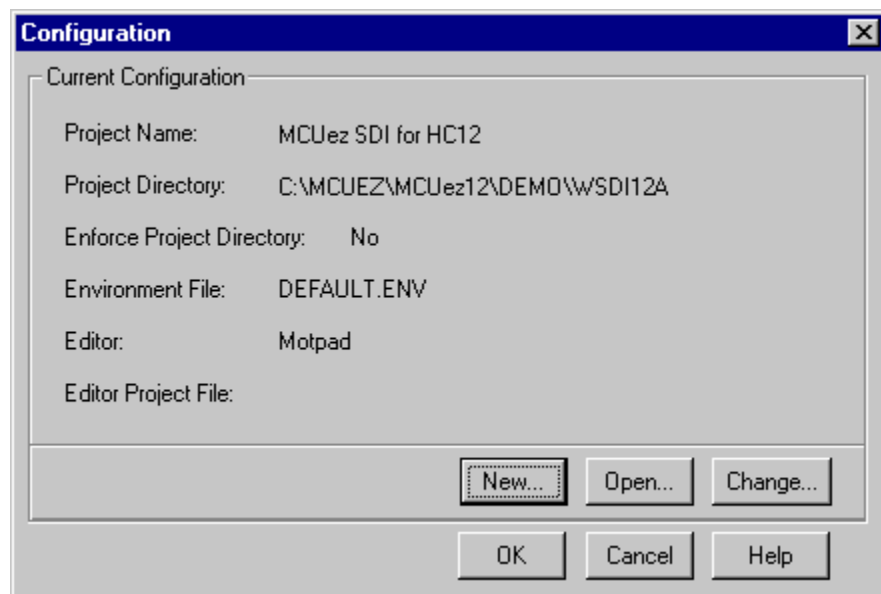
Figure 1-2. Environment Configuration Dialog Box

3. Click on the **New** button to open the **Project Directory** dialog box.

4. Enter the path for the new project in the edit box. For example, substitute `C:\MCUEZ\MCuez12\DEMO\WMMDS12A` with `C:\MCUEZ\MCUez12\DEMO\mydir` as the example shows in **Figure 1-3**.
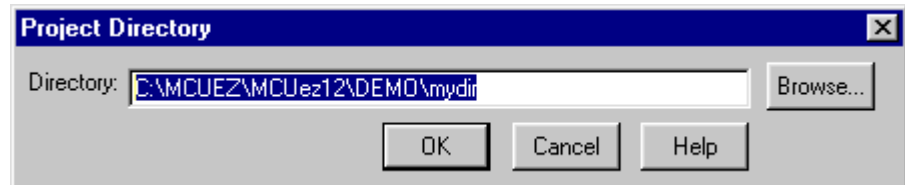
Figure 1-3. Working Project Directory Dialog Box

NOTE: *The specified directory must be accessible from a PC.*

5. Click on the **OK** button to close the **Project Directory** dialog box. The **New Configuration** dialog box will then appear.

6. Define the editor to use with the project. Select the **Editor** tab. Select an editor from the **Editor** drop down box. In the Executable command line, enter the path and command used to start the editor.

    For example:

    `C:\MCUEZ\MCUez12\Prog\Motpad.EXE`

    The command also can be selected by using the **Browse...** button.
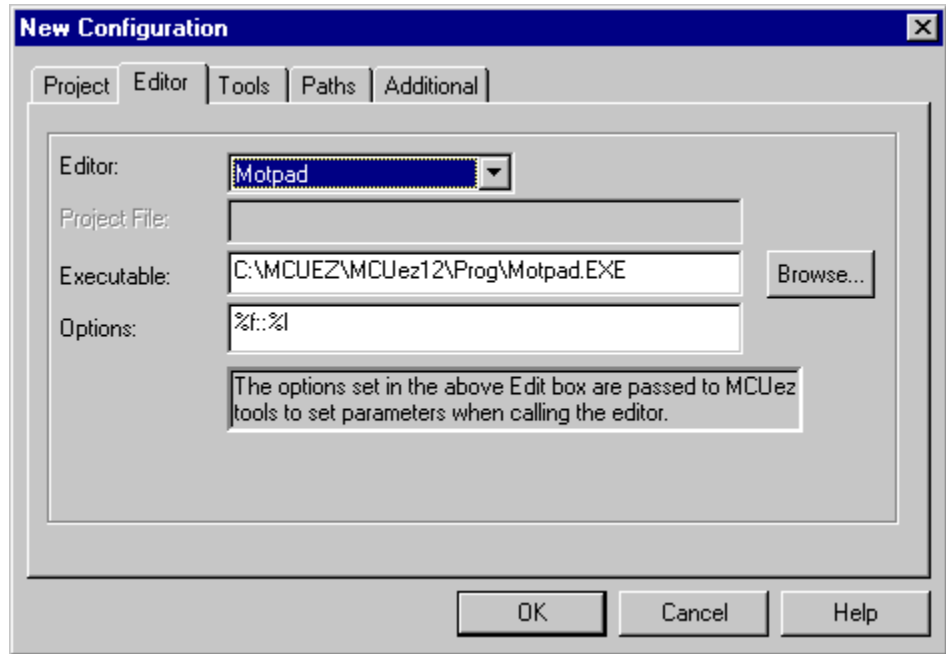
Figure 1-4. New Configuration Dialog Box

7. Click on the **OK** button in the **New Configuration** dialog box to create the MCUez configuration files in the specified project directory.

1.4.2 Creating an Assembly Source File

Once the project has been configured, writing the application can begin. For example, source code may be stored in a file named *test.asm* and may look as like this:

```
            XDEF entry      ; Make the symbol entry visible for
                            ; external module.
                            ; This is necessary to allow the
                            ; linker to find the symbol and
                            ; use it as the entry point for
                            ; the application.
initStk:    EQU $AFE        ; Initial value for SP
dataSec:    SECTION         ; Define a section
var1:       DC.W 5          ; Assign 5 to the symbol var1
codeSec:    SECTION         ; Define a section for code
entry:
    LDS #initStk            ; Load stack pointer
    LDD var1
    BRA entry
```

When writing assembly source code, pay special attention to these points:

- All symbols referenced outside the current source file (in another source file or in the linker configuration file) must be visible externally. For this reason, the assembly directive XDEF entry has been inserted.

- To make debugging from the application easier, defining separate sections for code, constant data (defined with DC (define constant)), and variables (defined with DS (define space)) are strongly recommended. This enables the symbols located in the variable or constant data sections to be displayed in the data window component of the debugger.

- The stack pointer must be initialized when using BSR (branch to subroutine) or JSR (jump to subroutine) instructions in an application.

1.4.3 Assembling a Source File

This procedure describes how to assemble a source file.

1. Start the assembler by clicking on the **ezASM** button in the **MCUez Shell**. Enter the name of the file to be assembled in the editable combo box, as shown in **Figure 1-5**.
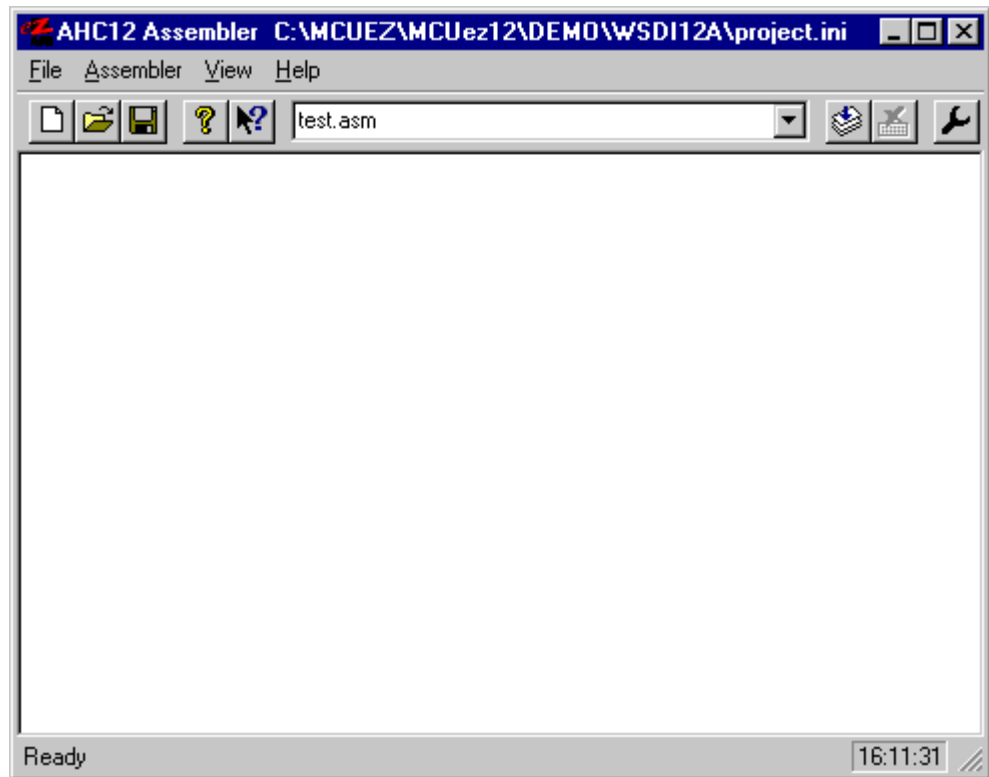
Figure 1-5. Assembler Window

2. Select the menu entry **Assembler | Options** to generate an *ELF/DWARF 2.0* object file. The **Options Settings** dialog is displayed as shown in **Figure 1-6**.
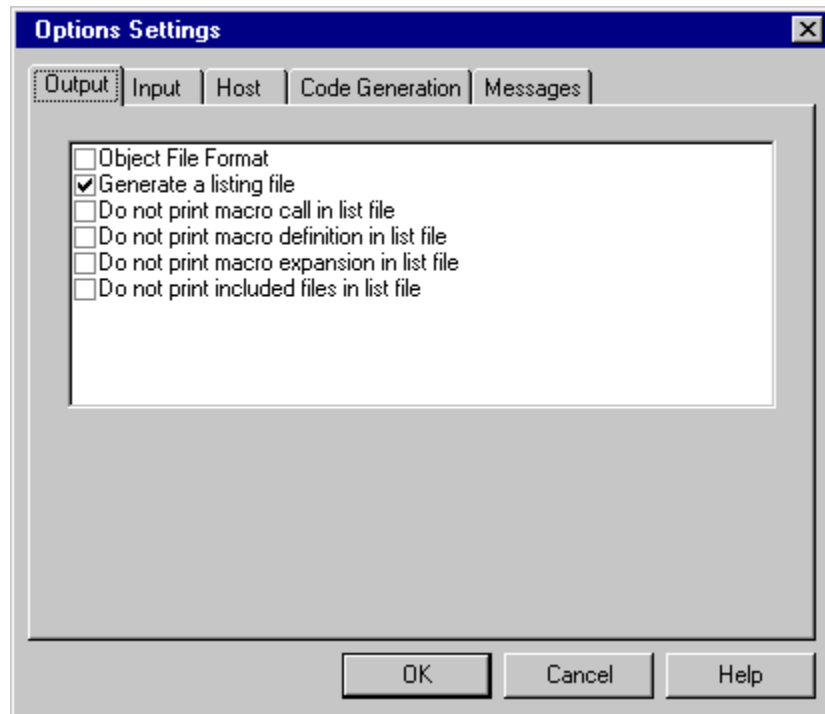
Figure 1-6. Options Settings Dialog Box

**For More Information On This Product,
Go to: www.freescale.com**

3. In the **Output** folder, select the check box in front of the label **Object File Format** shown in **Figure 1-7**. Select the radio button **ELF/DWARF 2.0 Object File Format** and click **OK**.
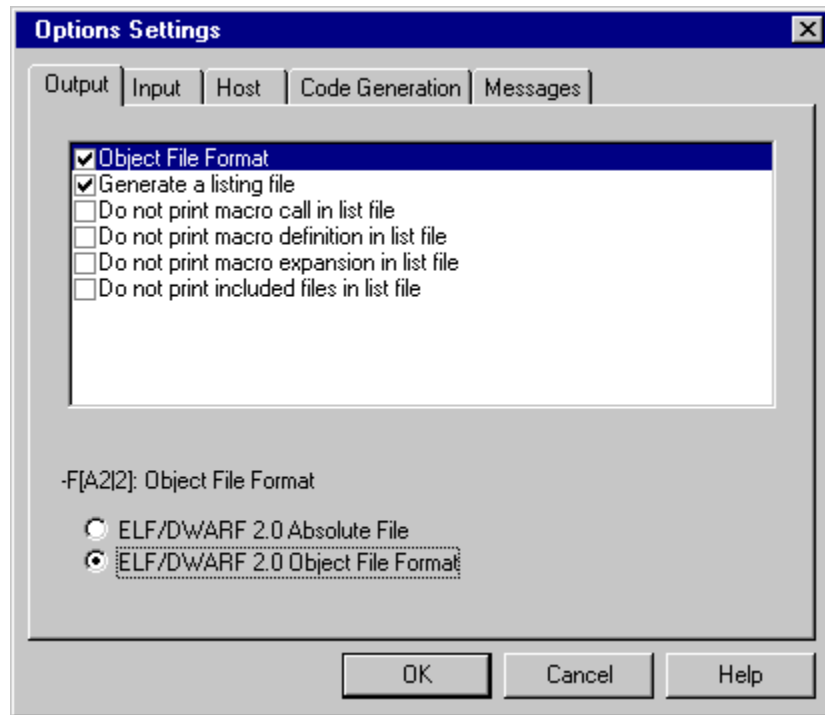
Figure 1-7. Selecting an Object File Format

4. The file is assembled, as shown in **Figure 1-8**, when the **Assemble** button is clicked.
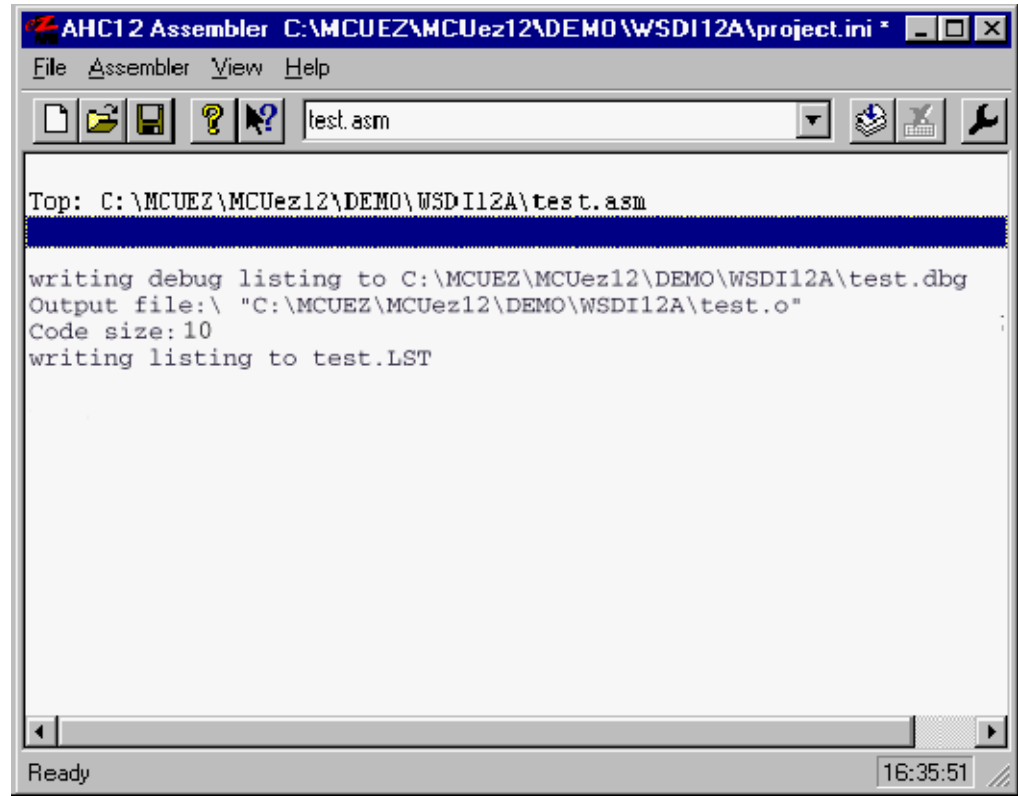
Figure 1-8. Assembling a File

The macro assembler indicates a successful assembler session by printing the number of generated bytes of code. The message Code size: 10 indicates that *test.asm* was assembled without errors. The macro assembler generates a binary object file and a debug listing file for each source file. The binary object file has the same name as the input module with an extension of *.o*. The debug listing file has the same name as the input module, with an extension of *.dbg*.

When the assembly option -L is specified on the command line, the macro assembler generates a list file containing the source instruction and corresponding hexadecimal code.

Freescale Semiconductor, Inc.

The list file generated by the macro assembler looks like this:

```
Motorola HC12-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997
Abs. Rel.   Loc.      Obj. code    Source line
---- ----   ------    ---------    -----------
1    1                             XDEF entry
...
4    4      0000      0AFE         initStk: EQU $AFE  ; SP Init
                                                      ; value
5    5                             dataSec: SECTION   ;
6    6      000000 0005            var1: DC.W 5       ; Assign 5 to
                                                      ; var1
7    7
     8                             codeSec: SECTION   ;
9    9                             entry:
10   10     000000 CF 0AFE         LDS #initStk       ; Load stack
11   11     000003 FC xxxx         LDD var1
12   12     000006 20F8            BRA entry
```

1.4.4 Linking an Application

Once the object file is available, the application can be linked. The linker will organize code and data sections according to the linker parameter file. Follow this procedure to link an application:

1. Start the editor and create the linker parameter file. Copy the file *fibo.prm* to *test.prm*.

2. In the file *test.prm*, change the name of the executable and object files to *test*.

3. Additionally, modify the start and end addresses for the ROM and RAM memory areas.

The *test.prm* module appears like this:

```
LINK test.abs           /* Name of the executable file generated.*/
NAMES test.o END        /*Name of the object files in the application*/

SEGMENTS
  MY_ROM = READ_ONLY  0x800 TO 0x8FF;        /*READ_ONLY memory area */
  MY_RAM = READ_WRITE 0xB00 TO 0xBFF;        /*READ_WRITE memory area */
END
PLACEMENT
  .data    INTO MY_RAM; /* Variables should be allocated in MY_RAM */
  .text    INTO MY_ROM; /* Code should be allocated in MY_ROM */
END
INIT    entry                  /* entry is the entry point to the application */
VECTOR ADDRESS 0xFFFE entry  /* Initialization for Reset vector */
```

NOTE: *The commands in the linker parameter file are described in detail in the MCUez Linker User's Manual, Motorola document order number MCUEZLNK/D.*

4. Click the **eZLink** button in the **MCUez Shell**. The linker is started as shown in **Figure 1-9**.

5. Enter the name of the file to be linked in the editable combo box. To start linking, press the **Enter** key or click on the **Link** button.

Link Button
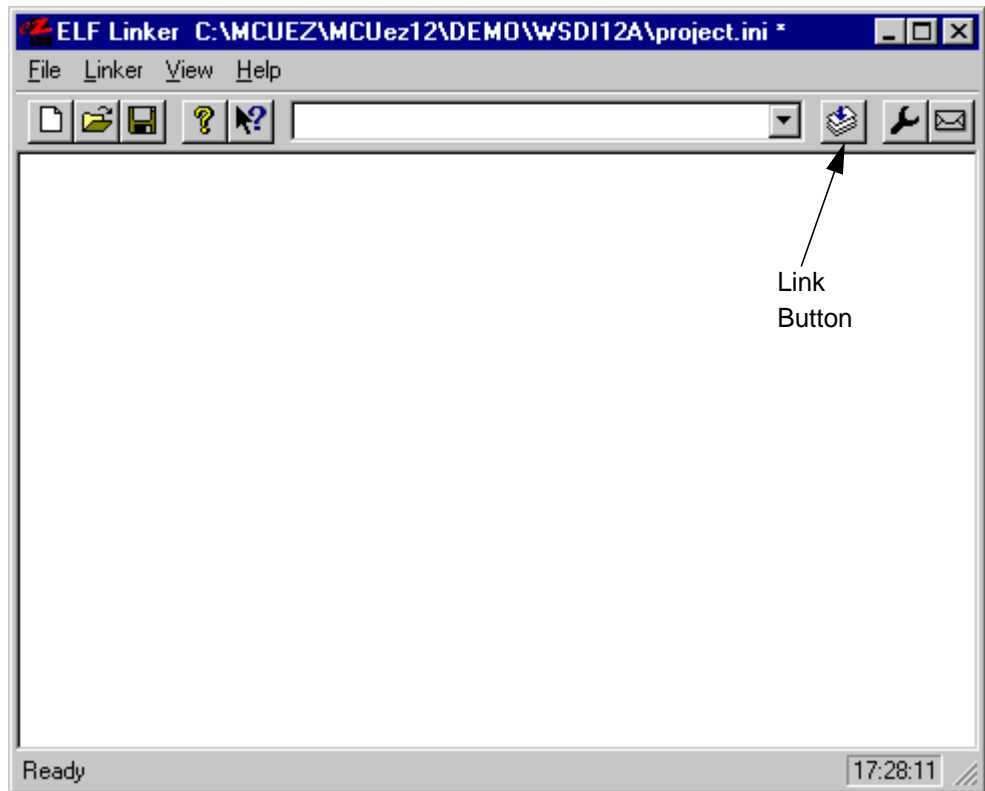
Figure 1-9. Linker Window

Once the linker is started, the linker window displays the link process as shown in **Figure 1-10**.
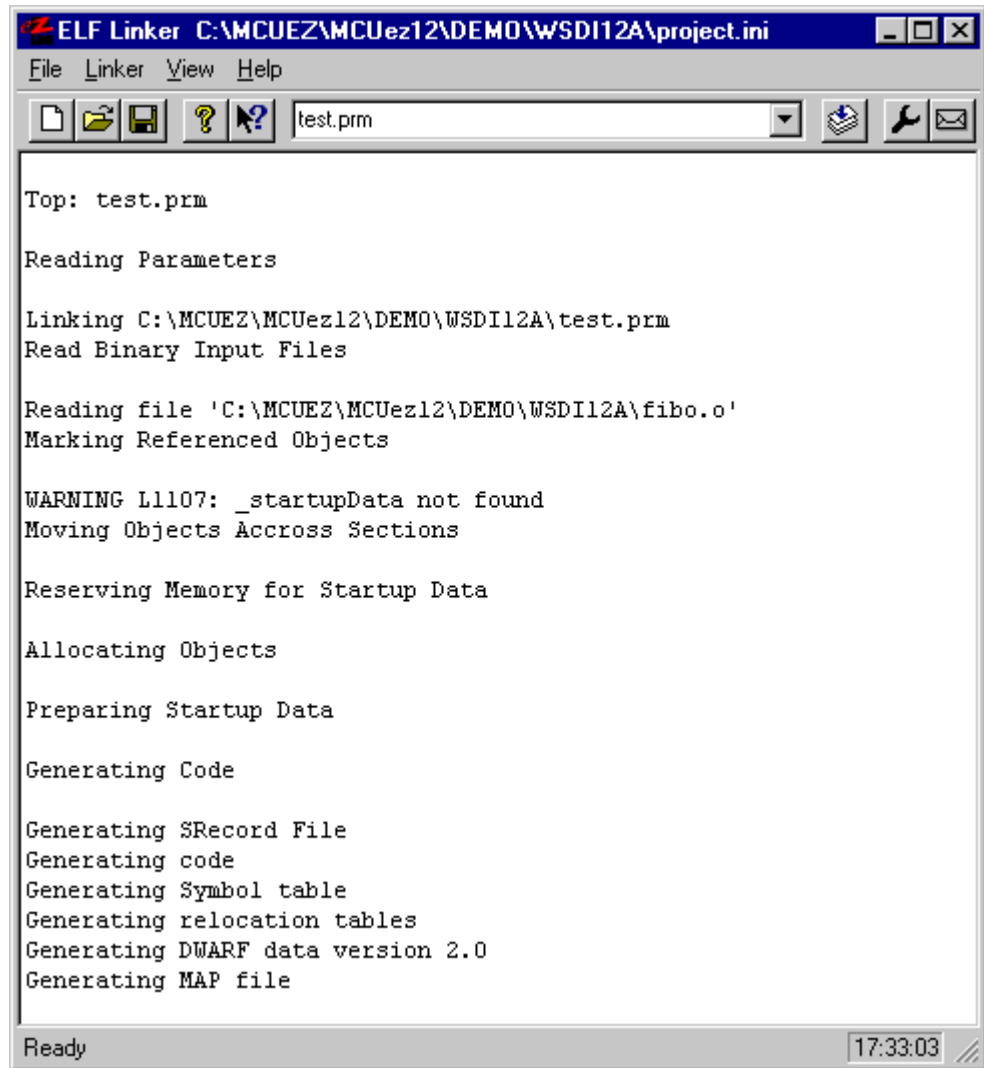
Figure 1-10. Link Process

Section 2. Graphical User Interface

2.1 Contents

2.2 Introduction

The MCUez HC12 assembler uses a Microsoft Windows$^{®}$ application, which is a graphical user interface (GUI).

2.3 Starting the Motorola Assembler

Start the assembler from the **MCUez Shell** by clicking on the **ezASM** icon in the toolbar.

When the assembler is started, a standard **Tip of the Day** window, containing tips about the assembler, is displayed.
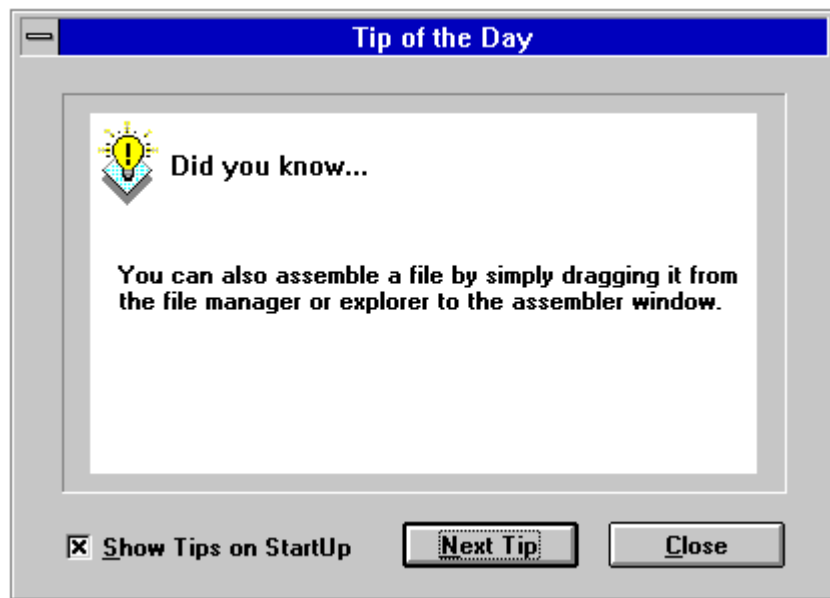
Figure 2-1. Tip of the Day Window

Click **Next Tip** to see the next piece of information about the assembler. Click **Close** to close the **Tip of the Day** dialog.

To bypass the standard **Tip of the Day** window when the assembler is started, uncheck *Show Tips on StartUp*.

To re-enable the tips window, choose the **Help|Tip of the Day ...** menu option. The **Tip of the Day** dialog will open. Then select **Show Tips on StartUp**.

2.4 Assembler Graphical Interface

If the assembler was started without specifying a filename, the window in **Figure 2-2** is displayed. The assembler window provides a window title, menu bar, toolbar, content area, and status bar.
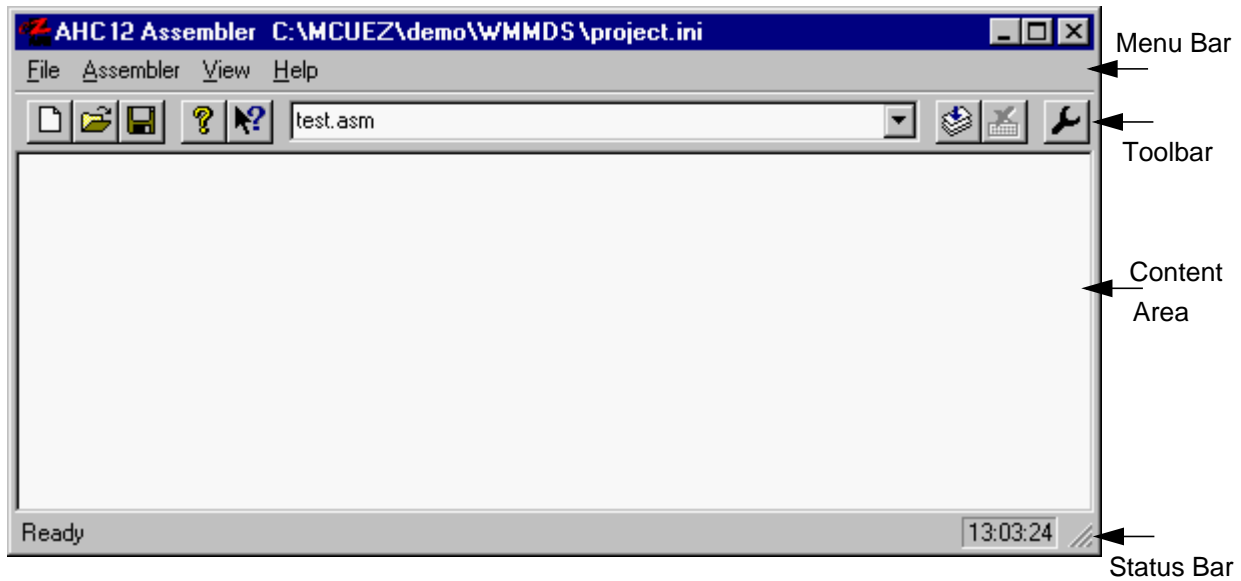
Figure 2-2. Assembler Window

2.4.1 Window Title

The window title displays the assembler name and project name. If no project is currently loaded, **Default Configuration** is displayed. An * (asterisk) after the project name indicates that some values have been changed. The * indicates changes in options, editor configuration, or appearance (window position, size, font, etc.).

2.4.2 Content Area

The content area displays logging information about the assembly session and consists of:

- Name of file being assembled

- Complete path and name of files processed (main assembly file and all included files)

- List of error, warning, and information messages

- Size of code generated during the assembly session

If a filename is dragged and dropped into the content area, the file is either loaded as a configuration file or is assembled. It is loaded as a configuration file if the file has a *.ini* extension. If not, the file is assembled with the current option settings. (See **2.4.8 Specifying the Input File**.)

Assembly information in the content area includes:

- Files created or modified

- Location within file where errors occurred

- A message number

Some files listed in the content area can be opened in the editor specified during project configuration. Double click on a filename to open an editable file or select a line that contains a filename and click the right mouse button to display a menu that contains an **Open ...** entry (if file is editable).

A message number is displayed with message output. From this output, there are three ways to open the corresponding help information.

1. Select one line of the message and press F1. Help for the associated message number is displayed. If the selected line does not have a message number, the main help is displayed.

2. Press Shift-F1 and then click on the message text. If there is no associated message number, the main help is displayed.

3. Click the right mouse button on the message text and select **Help on ...**. This menu entry is available only if a message number is available.

After an assembly session has completed, error feedback can be performed automatically by double clicking on the message in the content area. The source file containing the error or warning message will open to the line containing the problem.

2.4.3 Assembler Toolbar

Figure 2-3 illustrates the assembler toolbar.
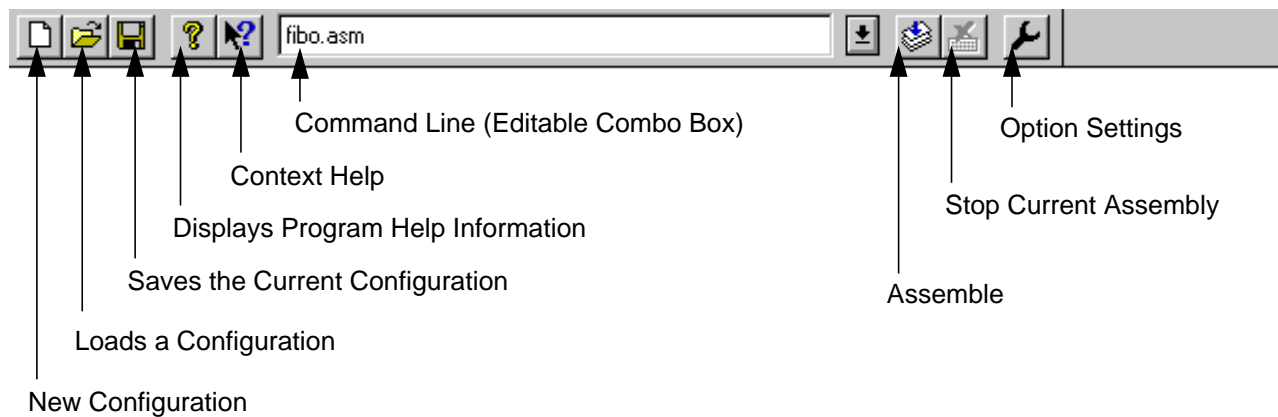
Figure 2-3. Assembler Toolbar

The three buttons on the left correspond with entries in the **File** menu. The **New Configuration**, **Load Configuration,** and **Save Configuration** buttons enable the user to reset, load, and save configuration files for the assembler.

The **Help** and **Context Help** buttons open the help file or use the context-sensitive help feature.

Press the **Context Help** button to change the mouse cursor to a question mark and arrow. Then click on an item within the application to display help information. Help is available for menus, toolbar buttons, and window areas.

The command line box contains a drop down list of the last commands executed. Once a command line has been selected or entered in the combo box, click the **Assemble** button to execute the command.

The **Options Setting** button opens the **Options Setting** dialog box.

2.4.4 Status Bar

Figure 2-4 shows the assembler status bar.

Figure 2-4. Assembler Status Bar

Point to a menu entry or button in the toolbar to display a brief explanation in the message area.

2.4.5 Assembler Menu Bar

The entries in **Table 2-1** are available in the **Menu Bar**.

Table 2-1. Menu Bar

Menu entry	Description
File	Assembler configuration file management
Assembler	Assembler option settings
View	Assembler window settings
Help	Standard windows help menu

2.4.6 File Menu

An assembler configuration file typically contains the following information:

- Assembler option settings specified in the assembler dialog boxes

- Last command line executed and current command line

- Window position, size, and font

- Editor associated with the assembler

- **Tip of the Day** settings

Assembler configuration information is stored in the specified configuration file. As many configuration files as required for a project can be defined. Switch to different configuration files by selecting **File|Load Configuration** and **File|Save Configuration**, or by clicking the corresponding toolbar buttons.

For instance:

- Choose **File|Assemble** to open a standard **Open File** dialog box. A list of all *.asm* files in the project directory is displayed. Select an input file. Click **OK** to close the dialog box and assemble the selected file.

- Choose **File|New/Default Configuration** to reset assembler options to the default values. Default values are specified in the section titled **Command Line Options**.

- Choose **File|Load Configuration** to open a standard **Open File** dialog box. A list of all *.ini* files in the project directory is displayed. Select a configuration file to be used by subsequent assembly sessions.

- Choose **File|Save Configuration** to store the current settings in the configuration file specified in the title bar.

- Choose **File|Save Configuration as ...** to open a standard **Save As** dialog box and display the list of all *.ini* files in the project directory. Specify the name and location of the configuration file. Click **OK** to save the current settings in the specified configuration file.

- Choose **File|Configuration ...** to open the **Configuration** dialog box. Specify an editor and related information to be used for error feedback, then save the configuration.

Freescale Semiconductor, Inc.

2.4.6.1 Editor Settings Dialog

This dialog box has several radio buttons for selecting a type of editor. Depending on the type selected, the content below it changes.

These are the main entries:

- **Global Editor (Configured by the Shell)**
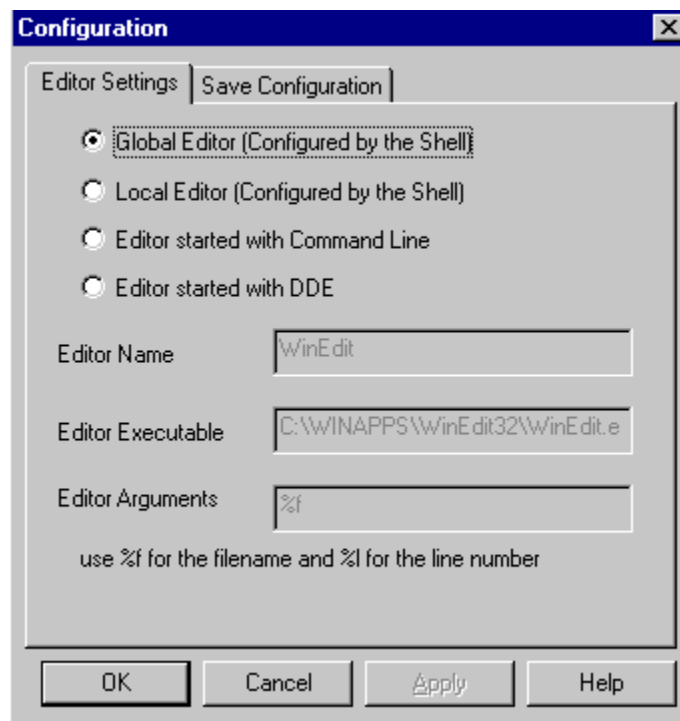
Figure 2-5. Starting the Global Editor

This entry is enabled only when an editor is defined in the **[Editor]** section of the global initialization file *mcutools.ini*.

- **Local Editor (Configured by the Shell)**
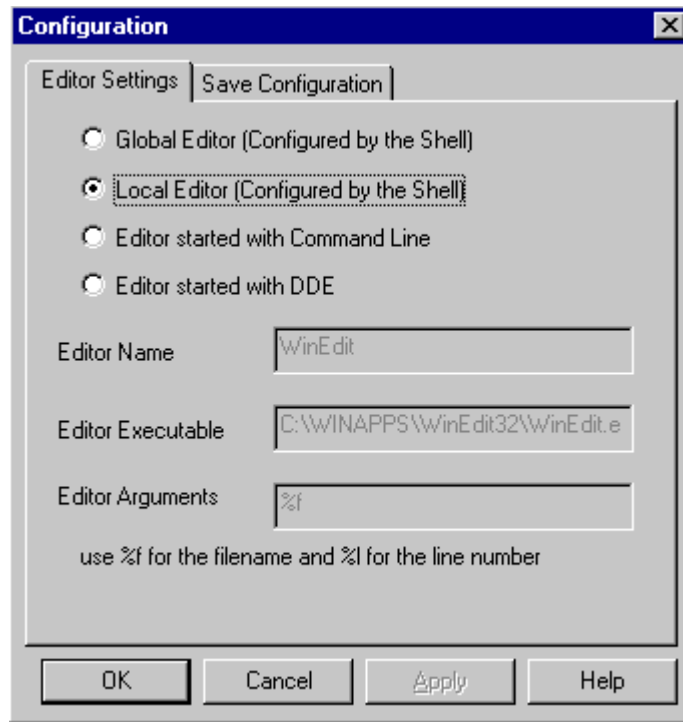
Figure 2-6. Starting the Local Editor

This entry is only enabled if an editor is defined in the local configuration file, usually *project.ini* in the project directory.

The **Global Editor** and **Local Editor** settings cannot be edited within this dialog box, since they are read only. These entries can be configured with the **MCUez Shell** application.

• **Editor started with Command Line**
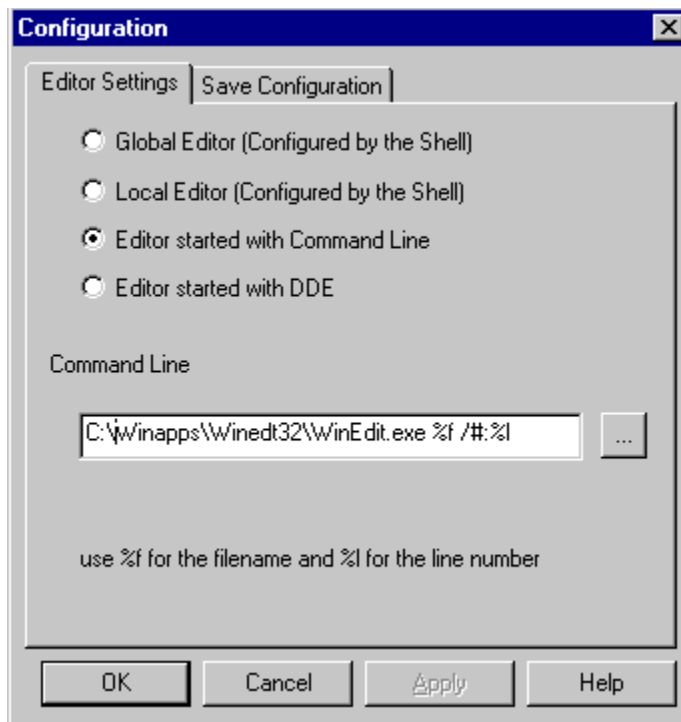
Figure 2-7. Starting the Editor with the Command Line

When this editor type is selected, a separate editor is associated with the assembler for error feedback. The editor configured in the shell will not be used for error feedback. Enter the appropriate path and command name to start the editor. Command modifiers are specified on the command line.

Example:

For WinEdit™ 32-bit version

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

For Write

    C:\Winnt\System32\Write.exe %f

    Write does not support line number modifier.

For Motpad

    C:\TOOLS\MOTPAD\MOTPAD.exe %f::%l

    Motpad supports line numbers.

- **Editor started with DDE**
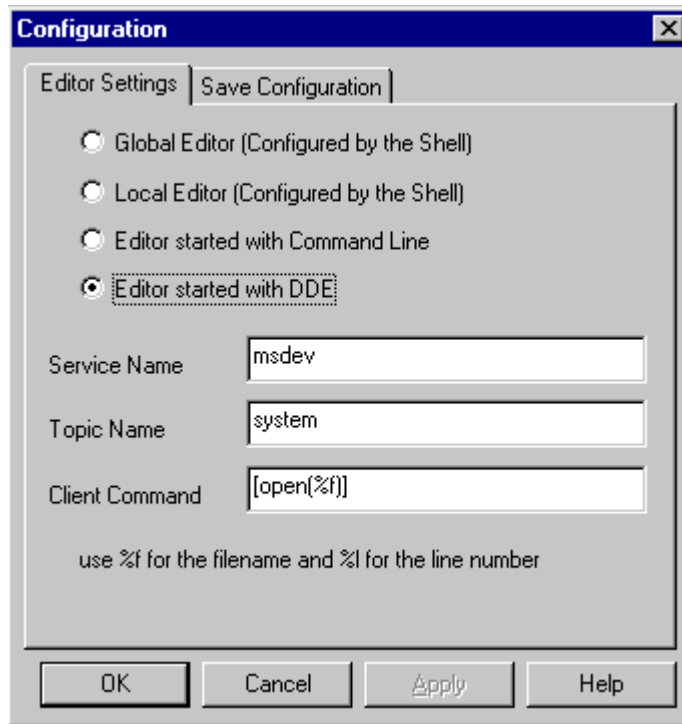
Figure 2-8. Starting the Editor with DDE

Enter the service, topic, and client name to be used for a DDE connection to the editor. All entries can have modifiers for filename and line number as explained in the next example.

Example: For Microsoft Developer Studio®, use this setting:

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

- Modifiers

  When either entry **Editor Started with the Command** line or **Editor started with DDE** is selected, the configuration may contain modifiers to identify which file to open and which line to select.

  – The `%f` modifier refers to the name of the file (including path) where the error has been detected.

  – The `%l` modifier refers to the line number where the message has been detected.

The editor format depends on the command syntax used to start the editor. Check the editor manual for modifiers that can be used to define the editor command line.

NOTE: *Be cautious when using the `%l` modifier. This modifier can be used only with an editor that can be started with a line number as a parameter. Editors such as WinEdit version 3.1 or lower and Notepad do not allow this kind of parameter.*

NOTE: *When using a word processing editor, such as Microsoft Word® or Wordpad, make sure to save the input file as an ASCII text file; otherwise, the assembler will have trouble processing the file.*

2.4.6.2 Save Configuration Dialog

Figure 2-9 shows the **Save Configuration** dialog box.
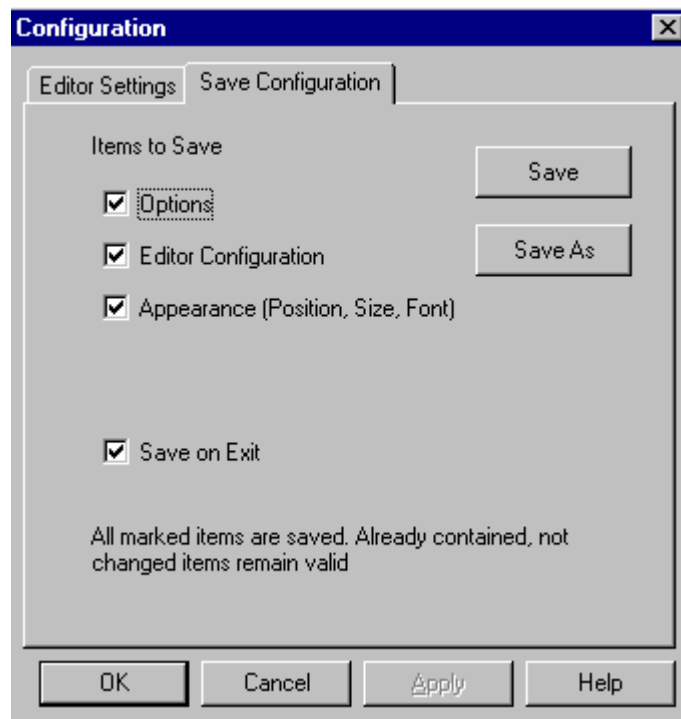
Figure 2-9. Save Configuration Dialog Box

The second page of the configuration dialog consists of save operations. In the **Save Configuration** dialog, select attributes to be stored in the project file. This dialog box provides the following configurations:

- **Options** — When set, the current option settings are stored in the configuration file. Disable this option to retain the last saved options.

- **Editor configuration** — When set, the current editor settings are stored in the configuration file. Disable this option to retain the last saved options.

- **Appearance**— When set, the current application appearance, such as the window position (only loaded at startup time) and the command line content and history, is saved. Disable to keep previous settings.

- **Save on exit** — If this option is set, the assembler will save the configuration on exit. No prompt will appear to confirm this operation. If this option is not set, the assembler will ignore any changes.

NOTE: *Almost all settings are stored in the configuration file. Exceptions are the recently used configuration list and all settings in this dialog. These settings are stored in the assembler section of the mcutools.ini file.*

Assembler configurations can coexist in the same file used for the project configuration (defined by the shell application) along with other MCUez tool specifications. When an editor is configured by the shell, the assembler can read this information from the project file, if present. The project configuration file created by the shell is named project.ini. Therefore, this filename is also suggested (but not mandatory) to the assembler.

2.4.6.3 Assembler Menu

Table 2-2 depicts the **Assembler** menu that allows customization of the assembler and setting or resetting of assembler options.

Table 2-2. Assembler Menu

Item	Description
Options	Allows defining of the options to be activated when assembling an input file

2.4.7 View Menu

This menu enables customization of the assembler window. For instance, whether the status bar or toolbar will be displayed or hidden can be defined. The user also can define the font used in the window or clear the window.

- Choose **View|Tool Bar** to switch on/off the assembler window toolbar.

- Choose **View|Status Bar** to switch on/off the assembler window status bar.

- Choose **View|Log ...** to customize the output in the assembler window content area.

- Choose **View|Log ...|Change Font** to open a standard **Font Selection** dialog box. Options selected in this dialog are applied to the assembler window content area.

- Choose **View|Log ...|Clear Log** to clear the assembler window content area.

2.4.7.1 Option Settings Dialog Box

This dialog box enables the user to set/reset assembler options, as shown in **Figure 2-10**.
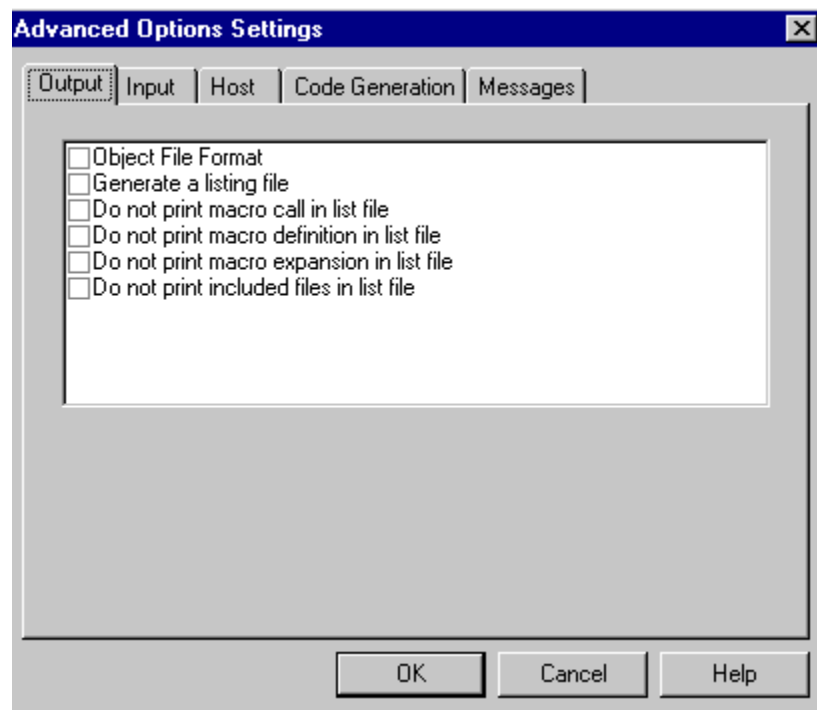
Figure 2-10. Option Settings Dialog Box

Available options are arranged in different groups as shown in **Table 2-3**.

Table 2-3. Advanced Options

Option Group	Description
Output	Lists options related to the output files generated (type of files to be generated)
Input	Lists options related to the input file
Host	Lists options related to the host
Code Generation	Lists options related to code generation (memory models, ...)
Messages	Lists options controlling the generation of error messages

An assembly option is set when the corresponding check box is checked. To obtain more information about a specific option, select the option and press the F1 key or the **Help** button. To select an option, click once on the option text.

NOTE: *Options that require additional parameters will display an edit box or an additional subwindow where additional parameters can be set.*

Assembler options specified in the project file (using the **MCUez Shell**) are automatically displayed in the **Option Settings** dialog box.

2.4.8 Specifying the Input File

The input file to be assembled can be specified in several ways. During the assembly session, options will be set according to the configuration provided by the user in the **Option Settings** dialog box. Before assembling a file, make sure a project directory is associated with the assembler.

2.4.8.1 Using the Editable Combo Box in the Toolbar

The following describes how to use the **Editable Combo** box.

- **Assembling a new file** — A new filename and additional assembler options can be entered on the command line. Click on the **Assemble** button or press the **Enter** key to assemble the specified file.

- **Reassembling a file** — The previously executed command can be displayed by clicking on the arrow on the right side of the command line. From the drop down list, select a command. Click on the **Assemble** button or press the **Enter** key to assemble the specified file.

2.4.8.2 Using the Entry File | Assembly ...

Select the menu entry **File | Assemble** to display the **File to Assemble** dialog box. Browse to and select the desired file. Click **Open** to assemble the selected file.

2.4.8.3 Using Drag and Drop

A filename can be dragged from an external program (for example, the **File Manager**) and dropped into the assembler window. The dropped file is assembled as soon as the mouse button is released in the assembler window. If the dragged file has the extension *.ini*, it is a configuration file and will be loaded and not assembled.

2.5 Error Feedback

After a source file has been assembled, the content area displays a list of all error or warning messages detected. The message format is:

>> <FileName>, line <line number>, col <column number>
pos <absolute position in file>

<Portion of code generating the problem>
<message class> <message number>: <Message string>

Example:

```
>> in "C:\DEMO\fiboerr.asm", line 76, col 20, pos 1932
BRA label
              ^
ERROR A1104: Undeclared user defined symbol: label
```

Errors can be corrected by using the editor defined during configuration. Editors such as WinEdit Version 95 (or higher) or Codewright from Premia Corporation can be started with a line number in the command line. If configured correctly, these editors are activated automatically by double clicking on an error message. The editor will open the file containing the error and position the cursor on the line with the error.

Editors like WinEdit Version 31 or lower, Notepad, or Wordpad cannot be started with a line number. These editors can be activated automatically by double clicking on a message. The editor will open the file containing the error. To locate the error, use the find or search feature of the editor. In the assembler content area, select the line containing the message class, number, and string and press CTRL+C to copy the message. Paste the message in the **Find** dialog box of the editor to search for the error.

Section 3. Environment Variables

3.1 Contents

3.2 Introduction

This section describes environment variables used by the MCUez assembler. Environment variables are set in the Paths or Additional tab of the MCUez shell New Configuration or Current Configuration dialog box. Refer to the *MCUez Installation and Configuration User's Manual*, Motorola document order number MCUEZINS/D. Environment variables that define paths (such as GENPATH, OBJPATH, ABSPATH, etc.) are used by the assembler and other MCUez applications.

3.3 Paths

Environment variables that contain paths indicate where to look for files. A path list is a list of directory names separated by semicolons or a directory name preceded by an asterisk. If a directory name is preceded by an asterisk (`*`), the programs recursively search the whole directory tree for a file, not just the given directory. Directories are searched in the order they appear in the path list.

Syntax:
```
DirSpec;DirSpec;DirSpec
*DirectoryName
```

Examples:
```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;

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

3.4 Line Continuation

It is possible to specify an environment variable over more than one line by using the line continuation character \ (back slash).

Example:

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

This is the same as

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

Observe the following when using the continuation character in path definitions:

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

Will result in

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

To avoid syntax errors, use a semicolon (;) at the end of a path if there is a \ at the end of the code line, such as:

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

3.5 Environment Variables Description

The remainder of this section describes each of the environment variables available for the assembler. The information in **Table 3-1** describes the structure for explaining each environment variable.

Table 3-1. Environment Variables

Topic	Description
Syntax	Specifies the syntax of the option in EBNF (Extended Backus-Naur Form) format
Arguments	Describes and lists optional and required arguments for the variable
Default	Shows the default setting for the variable, if applicable
Description	Provides a detailed description of the environment variable and how to use it
Example	Gives an example of usage and effects of the variable where possible
Tools	Lists tools that use this variable, if applicable
MCUez Shell	Explains how the environment variable can be initialized in the **MCUez Shell**
See also	Lists related sections, if applicable

3.5.1 ASMOPTIONS

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 they don't have to be specified each time a file is assembled.

Options listed here must be valid assembler options and are separated by space characters.

Example: `ASMOPTIONS=-W2 -L`

MCUez Shell: Open the **Current Configuration** dialog box.
Select the **Additional** tab.
Enter the environment variable definition in the edit box.

See also: **Section 5. Assembler Options**

3.5.2 GENPATH

Syntax: GENPATH=<path>

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

Description: The macro assembler will look for the source or included files first in the project directory, then in the directories listed in the environment variable GENPATH.

NOTE: *If a directory specification in this environment variable starts with an asterisk (*), the entire directory tree is searched recursively, for instance, all subdirectories are searched.*

Example: GENPATH=\sources\include;..\..\headers;*\user

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Paths** tab.
In the **Configure** combo box, select **General Path**.
Enter the directories in the list box (one directory on each line).

See also: None

3.5.3 ABSPATH

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 object
                files. When this environment variable is defined, the assembler
                will store the absolute files it produces in the first directory
                specified. 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

MCUez Shell: Open the **Current Configuration** dialog.
                Select the **Paths** tab.
                In the **Configure** combo box, select **Absolute**.
                Enter the directories in the list box (one directory on each line).

See also: None

3.5.4 OBJPATH

Syntax: `OBJPATH=<path>`

Arguments: \<path>: Paths separated by semicolons, without spaces

Description: When this environment variable is defined, the assembler will store the object files it produces in the first directory specified. If OBJPATH is not set, the generated object files will be stored in the directory where the source file was found.

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

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Paths** tab.
In the **Configure** combo box, select **Object**.
Enter the directories in the list box (one directory on each line).

See also: None

3.5.5 TEXTPATH

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. If TEXTPATH is not set, the generated listing files will be stored in the directory where the source file was found.

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

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Paths** tab.
In the **Configure** combo box, select **Text**.
Enter the directories in the list box (one directory on each line).

See also: None

3.5.6 SRECORD

Syntax: SRECORD=<RecordType>

Arguments: <Record Type>: Force the type for the Motorola S record that must be generated. This parameter may take the value S1, S2, or S3.

Description: This environment variable is only relevant when absolute files are generated directly by the macro assembler instead of object files. When this environment variable is defined, the assembler will generate a Motorola S 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).

When this variable is not set, the type of S record generated will depend on the size of the address loaded there. If the address can be coded on two bytes, an S1 record is generated. If the address is coded on three bytes, an S2 record is generated. Otherwise, an S3 record is generated.

NOTE: *If the environment variable SRECORD is set, it is the user's responsibility to specify the appropriate S record type. If S1 is specified while the code is loaded above 0xFFFF, the Motorola S file generated will not be correct because the addresses will all be truncated to 2-byte values.*

Example: SRECORD=S2

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable in the list box.

See also: None

For More Information On This Product,
Go to: www.freescale.com

3.5.7 ERRORFILE

Syntax: ERRORFILE=<filename>

Arguments: <filename>: Filename with possible format specifiers

Description: The environment variable ERRORFILE specifies the name of the error file used by the 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 (path included; same as %p%n)

Examples: ERRORFILE=MyErrors.err

Logs all errors in the file *MyErrors.err* in the current directory

ERRORFILE=\tmp\errors

Logs all errors in the filenamed *errors* in the directory *\tmp*

ERRORFILE=%f.err

Logs all errors in a file with the same name as the source file (with extension *.err*) into the same directory as the source file. For example, if the file *\sources\test.asm* is assembled, an error file *\sources\test.err* will be generated.

ERRORFILE=\dir1\%n.err

An error file *\dir1\test.err* will be generated for a source file named *test.asm*.

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

An error file *\dir1\dir2\errors.txt* will be generated for a source file *\dir1\dir2\test.asm*.

If the environment variable ERRORFILE is not set, errors are written to the default error file. The default error filename is dependent upon how the assembler is configured and started. If no filename is provided, errors are written to the *err.txt* file in the project directory.

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable definition in the list box.

See also: None

3.5.8 COPYRIGHT: Copyright Entry in Object File

Tools: Assembler and linker

Syntax: `COPYRIGHT=<copyright string>`

Arguments: `<copyright string>`: String for the copyright entry in the object file

Default: None

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

Example: `COPYRIGHT=Copyright by Motorola`

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable definition in the list box.

See also: Environment variable **3.5.9 INCLUDETIME: Create Time in Object File**

Environment variable **3.5.10 USERNAME: User Name in Object File**

3.5.9 INCLUDETIME: Create Time in Object File

Tools: Assembler and linker

Syntax: `INCLUDETIME=(ON | OFF)`

Arguments: ON: Include time information in object file

OFF: Do not include time information in 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 undesirable if 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 for date and time are "none" in the object file.

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

Example: `INCLUDETIME=OFF`

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable definition in the list box.

See also: Environment variable **3.5.8 COPYRIGHT: Copyright Entry in Object File**

Environment variable **3.5.10 USERNAME: User Name in Object File**

3.5.10 USERNAME: User Name in Object File

Tools:	Assembler and linker
Syntax:	USERNAME=<user>
Arguments:	<user>: Name of user
Default:	None
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 a decoder.
Example:	USERNAME=MOTOROLA
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Additional** tab. Enter the environment variable definition in the list box.
See also:	Environment variable **3.5.8 COPYRIGHT: Copyright Entry in Object File**
	Environment variable **3.5.9 INCLUDETIME: Create Time in Object File**

Freescale Semiconductor, Inc.

Section 4. Files

4.1 Contents

4.2 Introduction

This chapter describes all file types associated with the MCUez application.

4.3 Input Files

The following sections describe input files:

- Source files

- Include files

4.3.1 Source Files

The macro assembler takes any file as input and does not require the filename to have a special extension. However, it is suggested that all source filenames have the extension *.asm* and all included files have the extension *.inc*. Source files will be searched first in the project directory and then in the GENPATH directory.

4.3.2 Include Files

The search for include files is governed by the environment variable GENPATH. Include files are searched first in the project directory, then in the directories specified in the environment variable GENPATH. The project directory is set from the **MCUez Shell** or the environment variable DEFAULTDIR.

4.4 Output Files

The following sections describe six types of output files:

1. Object files

2. Absolute files

3. Motorola S files

4. Listing files

5. Debug listing files

6. Error listing files

4.4.1 Object Files

After a successful assembly 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 environment variable OBJPATH. If that variable contains more than one path, the object file is written to the first directory given. If this variable is not set, the object file is written to the directory where the source file was found. Object files always get the extension *.o*.

4.4.2 Absolute Files

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

4.4.3 Motorola S Files

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

For instance:

- If SRECORD = S1, the Motorola S record file gets the extension *.s1*.
- If SRECORD = S2, the Motorola S record file gets the extension *.s2*.
- If SRECORD = S3, the Motorola S record file gets the extension *.s3*.
- If SRECORD is not set, the Motorola S record file gets the extension *.sx*.

This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the motorola S file is written in the first directory given. If this variable is not set, the file is written in the directory the source file was found.

4.4.4 Listing Files

After a successful assembly session, the macro assembler generates a listing file containing each assembly instruction with its associated hexadecimal code. This file is generated when the option -L is activated, even if the macro assembler generates an absolute file. This file is written to the directory given in the environment variable TEXTPATH. If that variable contains more than one path, the listing file is written in the first directory specified. If this variable is not set, the listing file is written in the directory where the source file was found. Listing files always get the extension *.lst*. **Section 10. Assembler Listing File** describes the format of this file.

4.4.5 Debug Listing Files

After a 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 generates an absolute file. The debug listing file is a duplicate of the source, where all the macros are expanded and the include files merged. This file (with the extension *.dbg*) is written to the directory listed in the environment variable OBJPATH. If that variable contains more than one path, the debug listing file is written to the first directory given. If this variable is not set, the file is written in the directory where the source file was found. Debug listing files always get the extension *.dbg*.

4.4.6 Error Listing Files

If the macro assembler detects any errors, it creates an error file. The name and location of this file depend on the settings from the environment variable ERRORFILE.

If the macro assembler's window is open, it displays the full path of all include files read. After successful assembly, the number of code bytes generated and the number of global objects written to the object file are displayed. **Figure 4-1** shows the different structures associated with the assembler.
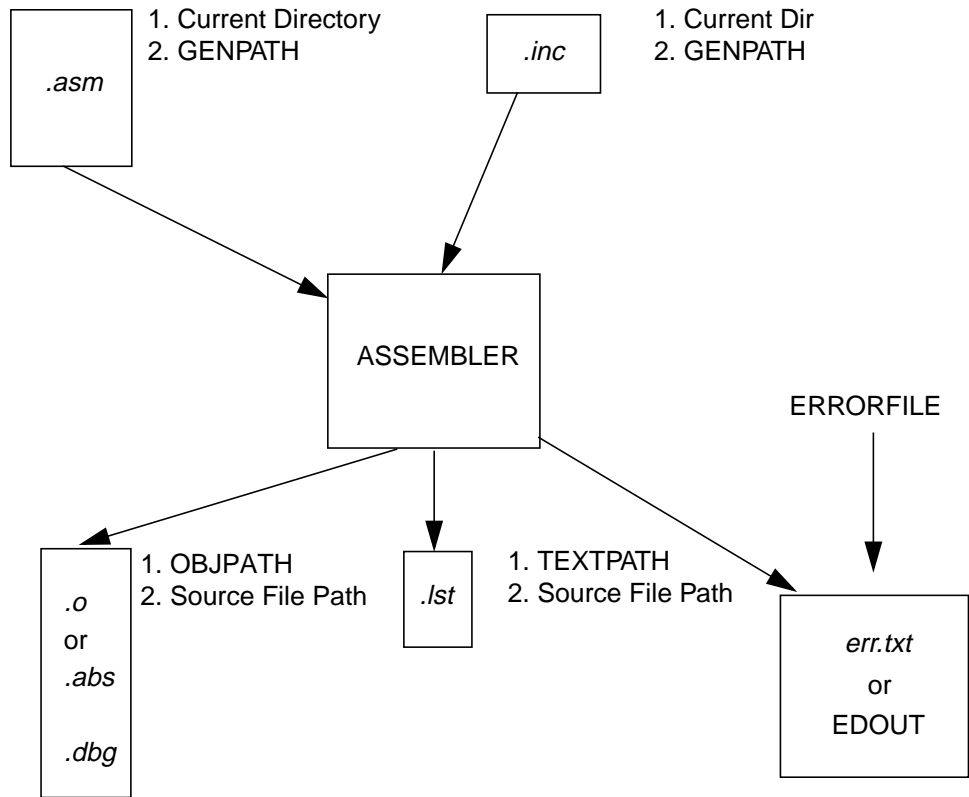
Figure 4-1. Assembler Structural Diagram

Section 5. Assembler Options

5.1 Contents

5.2 Introduction

The assembler offers a number of options that control how the assembler operates. Options consist of a dash (-) followed by one or more letters or digits. Anything not starting with a dash is assumed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the ASMOPTIONS environment variable. Typically, each assembler option is specified only once per assembly session.

NOTE: *Arguments for an option must not exceed 128 characters.*

Command line options are not case sensitive. $-Li$ is the same as $-li$. For options that belong to the same group, for example $-Lc$ and $-Li$, the assembler allows options to be combined, for example, $-Lci$ or $-Lic$ instead of $-Lc$ $-Li$.

NOTE: *It is not possible to combine options in different groups, for instance, $-Lc$ $-W1$ cannot be abbreviated by the terms $-LC1$ or $-LCW1$.*

5.3 ASMOPTIONS

If this environment variable is set, the assembler appends the values (options) defined for this variable 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 the user doesn't have to specify them each time a file is assembled.

5.4 Assembler Options

Table 5-1 describes how assembler options are grouped and Table 5-2 describes the scope of each option.

Table 5-1. Assembler Option Group

Group	Description
HOST	Lists options related to the host
OUTPUT	Lists options related to output file generation (type of file to be generated)
INPUT	Lists options related to input file
CODE	Lists options related to code generation (memory models, float format, etc.)
MESSAGE	Lists options controlling generation of error messages
VARIOUS	Lists various options

Table 5-2. Scope of Each Option

Scope	Description
Application	The 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	The option can be set differently for each assembly unit of an application. Mixing objects in an application is possible.
None	The option is not related to a specific code part. A typical example is options for message management.

Available options are arranged in separate groups, and a dialog box tab is available for each group. The content of the list box depends on the tab selected in the dialog box.

The remainder of this section describes each of the options available for the assembler. The options are listed in alphabetical order and described by the categories shown in Table 5-3.

Table 5-3. Assembler Option Details

Topic	Description
Group	HOST, OUTPUT, INPUT, CODE, MESSAGE, VARIOUS
Scope	Application, assembly unit, or none
Syntax	Specifies the syntax of the option in 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.
See also	Related options

5.4.1 -CI

-CI:	Set case sensitivity for label names OFF
Group:	INPUT
Scope:	Assembly unit
Syntax:	-CI
Arguments:	None
Default:	ON

Description: Switches case sensitivity OFF for label names. When this option is activated, the assembler ignores case sensitivity for label names.

This option can be only specified when the assembler generates an absolute file. (Option -FA2 must be activated.)

Example: When case sensitivity for label names is switched off, the assembler will not generate error messages for this code:

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

The instruction BRA Entry will branch on the label entry. By default, the assembler is case sensitive for label names. The labels Entry and entry are two distinct labels.

See also: None

Assembler Options

5.4.2 -Env

-Env:	Set environmental variable
Group:	HOST
Scope:	Assembly unit
Syntax:	`-Env` <EnvironmentVariable>=<VariableSetting>
Arguments:	<EnvironmentVariable>: Environment variable to be set <VariableSetting>: Assigned value
Default:	None
Description:	This option sets an environment variable.
Example:	`ASMOPTIONS=-EnvOBJPATH=\sources\obj`
	This is the same as `OBJPATH=\sources\obj` in the *default.env* file.
See also:	**Section 3. Environment Variables**

5.4.3 -F2 -FA2

-F:	Object file format
Group:	OUTPUT
Scope:	Application
Syntax:	-F (2 | A2)
Arguments:	2: *ELF/DWARF 2.0* object file format
	A2: *ELF/DWARF 2.0* absolute file format (default)
Default:	-FA2
Description:	Defines format for the output file generated by the assembler
	With the option -F2 set, the assembler produces an *ELF/DWARF 2.0* object file.
	With the option -FA2 set, the assembler produces an *ELF/DWARF 2.0* absolute file.
Example:	ASMOPTIONS=-F2
See also:	None

5.4.4 -H

-H: Short help

Scope: None

Syntax: -H

Arguments: None

Default: None

Description: The -H option will display a short list of available options.

                No other option or source file should be specified when the -H option is invoked.

Example: The following is a portion of the list produced by the option -H:

```
MESSAGE:
-N     Show Notification box in case of
errors
-W1    Don't print INFORMATION messages
-W2    Don't print INFORMATION or WARNING
messages

VARIOUS:
-H     Prints this list of options
-V     Prints the Assembler version
```

See also: None

5.4.5 -L

-L:	Generates a listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	-L
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file. This listing file will have the same name as the source file, but with the extension *.lst*. The listing file contains macro definitions, invocation, and expansion lines as well as expanded include files.
Example:	ASMOPTIONS=-L

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

When option -L is specified, the following portion of code:

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

Generates the following output in the assembly listing file:

```
         5    5                      INCLUDE
"macro.inc"
         6    1i                cpChar: MACRO
         7    2i                       LDD \1
         8    3i                       STD \2
         9    4i                       ENDM
        10    5i
        11    6                codeSec:
SECTION
        12    7                Start:
        13    8                       cpChar
ch1, ch2
        14    2m  000000 FC xxxx    +      LDD
ch1
        15    3m  000003 7C xxxx    +      STD
ch2
        16    9   000006 A7                NOP
        17   10   000007 A7                NOP
```

Contents of included files, as well as macro definition, invocation and expansion are stored in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.6 -Lc**, **5.4.7 -Ld**, **5.4.8 -Le**, and **5.4.9 -Li**

5.4.6 -Lc

-Lc:	No macro call in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	-Lc
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro invocations are not present in the listing file. The listing file contains macro definitions and expansion lines as well as expanded include files.
Example:	ASMOPTIONS=-Lc

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

When option -Lc is specified, the following portion of code:

```
cpChar: MACRO
          LDD \1
          STD \2
        ENDM
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
        NOP
```

Generates the following output in the assembly listing file:

```
          5     5                        cpChar:
MACRO
          6     6
LDD \1
          7     7
STD \2
          8     8
ENDM
          9     9                        codeSec:
SECTION
         10    10                         Start:
         12    6m   000000 FC xxxx    +
LDD char1
         13    7m   000003 7C xxxx    +
STD char2
         14    12    000006 A7
NOP
         15    13    000007 A7
NOP
```

Contents of included files, macro definitions, and expansion are stored in the list file. The source line containing the macro call is not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.7 -Ld

-Ld:	No macro definition in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	`-Ld`
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro definitions are not present in the listing file. The listing file contains macro invocation and expansion lines as well as expanded include files.

Example: `ASMOPTIONS=-Ld`

In the following example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one. When option `-Ld` is specified, the following portion of code:

```
cpChar: MACRO
            LDD \1
            STD \2
        ENDM
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
        NOP
```

Generates this output in the assembly listing file:

```
                 5     5                        cpChar:
MACRO
                 9     9                        codeSec:
SECTION
                10    10                          Start:
                11    11
cpChar char1, char2
                12     6m  000000 FC xxxx    +
LDD char1
                13     7m  000003 7C xxxx    +
STD char2
                14    12   000006 A7
NOP
                15    13   000007 A7
NOP
```

Contents of included files, as well as macro invocation and expansion are stored in the listing file. Source code from the macro definition is not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.8 -Le

-Le:	No macro expansion in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	-Le
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definitions and invocation lines as well as expanded include files.
Example:	ASMOPTIONS=-Le

In the following example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one. When option -Le is specified, the following portion of code:

```
cpChar: MACRO
            LDD \1
            STD \2
        ENDM
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
        NOP
```

Generates this output in the assembly listing file:

```
            5     5                                  cpChar:
MACRO
            6     6
LDD \1
            7     7
STD \2
            8     8
ENDM
            9     9                                  codeSec:
SECTION
           10    10                                    Start:
           11    11
cpChar char1, char2
           14    12   000006 A7
NOP
           15    13   000007 A7
NOP
```

Contents of included files, as well as macro definitions and invocation are stored in the listing file. Macro expansion lines are not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.9 -Li

-Li: No included file in listing file

Group: OUTPUT

Scope: Assembly unit

Syntax: -Li

Arguments: None

Default: None

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

Example: ASMOPTIONS=-Li

When option -Li is specified, this portion of code:

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

Generates the following output in the assembly listing file:

```
                5    5
INCLUDE "macro.inc"
                12   6                        codeSec:
SECTION
                13   7                          Start:
                15   3m  000000 FC xxxx   +
LDD char1
                16   4m  000003 7C xxxx   +
STD char2
                17   9   000006 A7
NOP
                18   10  000007 A7
NOP
```

Macro definition, invocation, and expansion are stored in the listing file.

See also: **5.4.5 -L**

5.4.10 -Ms -Mb

-M:	Memory model
Group:	CODE
Scope:	Application
Syntax:	$-M$ (s | b)
Arguments:	s: Small memory model b: Banked memory model
Default:	-Ms
Description:	The assembler for the MC68HC12 supports two different memory models. Default is the small memory model, which corresponds to the normal setup, for example, a 64-Kbyte code-address space. If a code memory expansion scheme is used, the banked memory model may be changed.
	Memory models should be observed when mixing ANSI C and assembler files. For compatibility reasons, the memory model used by the different files must be the same. Additionally, when assembling in the small memory model, the linker will check if all variables or code sections are located on the first page between 0 and FFFF.
Example:	ASMOPTIONS=-Ms
See also:	None

5.4.11 -MCUasm

-MCUasm:	Switch ON MCUasm compatibility
Group:	VARIOUS
Scope:	Assembly unit
Syntax:	`-MCUasm`
Arguments:	None
Default:	None
Description:	Switches ON MCUasm assembler compatibility mode. Additional features supported are listed in **Appendix B. MCUasm Compatibility**.
Example:	`ASMOPTIONS=-MCUasm`

5.4.12 -N

-N:	Display error notification box
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-N`
Arguments:	None
Default:	None
Description:	Causes the assembler to display an alert box if an error occurs during assembly. This is useful when running a makefile, since the assembler waits for the user to acknowledge the message, thus suspending makefile processing.
Example:	`ASMOPTIONS=-N`
	If an error occurs during assembly, a dialog box is displayed indicating the file where the error occurred.
See also:	None

Freescale Semiconductor, Inc.

5.4.13 -V

-V: Displays the assembler version

Group: VARIOUS

Scope: None

Syntax: -V

Arguments: None

Default: None

Description: Prints the assembler version and the current directory

NOTE: *This option is useful to determine the current directory.*

Example: -V   produces this list:
           Directory: C:\MCUEZ\demo\WMMDS12A
           Limitation Status: none
           Common Module V-5.0.4, Date Mar 18 1998
           Assembler Kernel, V-5.0.9, Date Mar 20 1998
           User Interface Module, V-5.0.14, Date Mar 18 1998
           Assembler Target, V-5.0.13, Date Mar 20 1998

See also: None

5.4.14 -W1

-W1:	No information messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-W1`
Arguments:	None
Default:	None
Description:	INFORMATION messages are not displayed. Only WARNING and ERROR messages are listed.
Example:	`ASMOPTIONS=-W1`
See also:	None

5.4.15 -W2

-W2:	No information and warning messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-W2`
Arguments:	None
Default:	None
Description:	INFORMATION and WARNING messages are not displayed. Only ERROR messages are listed.
Example:	`ASMOPTIONS=-W2`
See also:	None

Freescale Semiconductor, Inc.

5.4.16 -WmsgNe

-WmsgNe:	Number of error messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	-WmsgNe <number>
Arguments:	<number>: Maximum number of error messages
Default:	50
Description:	Sets the number of errors detected before the assembler stops processing
Example:	ASMOPTIONS=-WmsgNe2
	The assembler stops assembling after two error messages.
See also:	**5.4.17 -WmsgNi** and **5.4.18 -WmsgNw**

Freescale Semiconductor, Inc.

5.4.17 -WmsgNi

-WmsgNi:	Number of information messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-WmsgNi <number>`
Arguments:	\<number>: Maximum number of information messages
Default:	50
Description:	Sets the maximum number of information messages to be logged
Example:	`ASMOPTIONS=-WmsgNi10`
	The first 10 information messages are logged.
See also:	**5.4.16 -WmsgNe** and **5.4.18 -WmsgNw**

5.4.18 -WmsgNw

-WmsgNw:	Number of warning messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-WmsgNw  <number>`
Arguments:	<number>: Maximum number of warning messages
Default:	50
Description:	Sets the maximum number of warning messages to be logged
Example:	`ASMOPTIONS=-WmsgNw15`
	The first 15 warning messages are logged.
See also:	**5.4.16 -WmsgNe** and **5.4.17 -WmsgNi**

5.4.19 -WmsgFbv -WmsgFbm

-WmsgFb: Set message file format for batch mode

Group: MESSAGE

Scope: Assembly unit

Syntax: -WmsgFb [v | m]

Arguments: v: Verbose format

                m: Microsoft format

Default: -WmsgFbm

Description: The assembler can be started with additional arguments (for
                example, files to be assembled together with assembler
                options). If the assembler has been started with arguments (for
                example, from the **Make** tool or with the %f argument from
                WinEdit), the assembler assembles the files in batch mode. No
                assembler window is visible, and the assembler terminates
                after job completion.

                If the assembler is in batch mode, assembler messages are
                written to a file instead of to the screen. This file only contains
                the assembler messages. By default, the assembler uses a
                Microsoft message format to write the messages (errors,
                warnings, and information messages).

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

                Example:

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

By default, the assembler generates the following error information if it is running in batch mode:

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

Setting the format to verbose, more information is listed:

```
ASMOPTIONS=-WmsgFbv
>> in "X:\TW2.ASM", line 12, col 0, pos 215
        endif
        endif
^
ERROR A1001: Conditional else not allowed
here
```

See also: **5.4.20 -WmsgFiv -WmsgFim**

5.4.20 -WmsgFiv   -WmsgFim

-WmsgFi:	Set message file format for interactive mode
Group:	MESSAGE
Scope:	Assembly unit
Syntax:	-WmsgFi [v	m]
Arguments:	v: Verbose format
	m: Microsoft format
Default:	-WmsgFiv
Description:	If the assembler is started without additional arguments, the assembler is in interactive mode (a window is visible). By default, the assembler uses the verbose error file format to write the assembler messages (errors, warnings, and information messages). With this option, the default format may be changed from the verbose format (with source, line, and column information) to the Microsoft format (only line information).

NOTE: *Using the Microsoft format speeds up assembly, since the assembler writes less information to the screen.*

Example:

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

By default, the assembler generates the following error output in the assembler window if it is running in 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:

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

See also: **5.4.19 -WmsgFbv -WmsgFbm**

Section 6. Sections

6.1 Contents

6.2 Introduction

Sections are portions of code or data that cannot be split into smaller elements. Each section has a name, type, and attributes. Each assembly source file contains at least one section.

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

Sections with the same name, but from different modules, are combined into a single section when linked.

6.3 Section Attributes

According to content, an attribute is associated with each section. A section may be a:

- Code section

- Constant data section

- Data section

6.3.1 Code Sections

A section containing at least an instruction is considered to be a code section. Code sections are always allocated in the target processor ROM area. Code sections should not contain any variable definitions (variables defined using the DS (define space) directive). There is no write access on variables defined in a code section. Additionally, these variables cannot be displayed in the debugger as data.

Definitions are possible with self-modifiable code. The restriction using this process is that labels appearing in front of the DS directive will not appear in the data window.

6.3.2 Constant Data Sections

A section containing only constant data definitions (variables defined using the DC (define constant) or DCB (define constant block) directives) is considered to be a constant section. Constant sections should be allocated in the target processor ROM area; otherwise, they cannot be initialized when the application is loaded.

NOTE: *It is strongly recommended that the user defines separate sections for definitions of variables and constant variables. This will avoid any problems in the initialization of constant variables.*

6.3.3 Data Sections

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

Empty sections that do not contain any code or data declarations are also considered to be data sections.

6.4 Section Types

First, in an application, a programmer must decide which type of code to use:

- Absolute

- Relocatable

The assembler allows 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.

6.4.1 Absolute Sections

The starting address of an absolute section is known at assembly time. An absolute section is defined by the directive ORG. The operand specified in the ORG directive determines the start address, as shown in **Figure 6-1**.

```
        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           ; Load value in cst1
        ADDA cst2           ; Add value in cst2
        STAA var            ; Store in var
        BRA entry
```

Figure 6-1. Absolute Section Programming Example

In the previous example, two bytes of storage are allocated starting at address $A00. Symbol cst1 will be allocated at address $A00 and cst2 will be allocated at address $A01. All subsequent instructions or data allocation directives will be located in the absolute section until another section is specified using the ORG or SECTION directive.

When using absolute sections, the user is responsible for ensuring that no overlap exists 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 $200 bytes; otherwise, the sections starting at $A00 and $C00 will overlap.

When object files are generated, applications containing only absolute sections must be linked. In that case, there should be no overlap between address ranges from the absolute sections defined in the assembly file and address ranges defined in the linker parameter file.

The PRM (parameter) file used to link the previous example, is defined in **Figure 6-2**.

```
LINK test.abs          /* Name of the executable file generated. */
NAMES
  test.o               /* Name of object files in the application. */
END
SEGMENTS
/* 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  0x1000 TO 0x1FFF;
/* 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. */
  .data    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  .text    INTO MY_ROM;
END
 INIT entry            /*  Application entry point */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector */
```

Figure 6-2. PRM File Example Code

The linker PRM file contains at least:

- The name of the absolute file (command LINK)

- The name of the object file that should be linked (command NAMES)

- Specification of a memory area where the sections containing variables must be allocated. At least the predefined section .data must be placed there (command SEGMENTS and PLACEMENT). For applications containing only absolute sections, nothing will be allocated.

- Specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section .text must be placed there. For applications containing only absolute sections, nothing will be allocated.

- The application entry point (command INIT)

- Definition of the reset vector (command VECTOR ADDRESS)

6.4.2 Relocatable Sections

The start address of a relocatable section is evaluated at link time, according to the information stored in the linker parameter file. A relocatable section is defined through the directive SECTION, as illustrated in **Figure 6-3**.

```
        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      ; Load value in cst1
       ADDA cst2      ; Add value in cst2
       STAA var       ; Store in var
       BRA entry
```

Figure 6-3. Relocatable Section Programming Example

In the previous example, two bytes of storage are allocated in section `constSec`. Symbol `cst1` will be allocated at offset 0 and `cst2` at offset 1 from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable section `constSec` until another section is specified using the ORG or SECTION directive.

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

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

When all constant and code sections as well as data sections can be allocated consecutively, the PRM file used to assemble the example in **Figure 6-4** can be defined as follows:

```
LINK test.abs                  /* Name of the executable file generated. */
NAMES
  test.o                       /* Name of the object files in the application. */
END
SEGMENTS
/* 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. */
  .data    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  .text    INTO MY_ROM;
END
INIT entry                     /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

Figure 6-4. Defining One RAM and One ROM Area

The linker PRM file contains at least:

- The name of the absolute file (command LINK)

- The name of the object file which should be linked (command NAMES)

- Specification of a memory area where the sections containing variables must be allocated. At least the predefined section `.data` must be placed there (command SEGMENTS and PLACEMENT).

- Specification of a memory area where the sections containing code or constants must be allocated. At least the predefined section `.text` must be placed there (command SEGMENTS and PLACEMENT).

- Specification of application entry point (command INIT)

- Definition of the reset vector (command VECTOR ADDRESS)

According to the PRM file in **Figure 6-4**:

- The section `dataSec` will be allocated starting at `0x0800`.

- The section `constSec` will be allocated starting at `0x0B00`.

- The section `codeSec` will be allocated next to the section `constSec`.

When the constant, code, and data sections cannot be allocated consecutively, the PRM file used to link the previous example can be defined like this:

```
LINK test.abs          /* Name of the executable file generated. */
NAMES
  test.o               /* Name of the object files in the application. */
END
SEGMENTS
  ROM_AREA_1= READ_ONLY  0xB00 TO 0xB7F; /* READ_ONLY memory area. */
  ROM_AREA_2= READ_ONLY  0xC00 TO 0xC7F;/* READ_ONLY memory area. */
  RAM_AREA_1= READ_WRITE 0x800 TO 0x87F;/* READ_WRITE memory area. */
  RAM_AREA_2= READ_WRITE 0x900 TO 0x97F;/* READ_WRITE memory area. */
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec          INTO RAM_AREA_2;
  .data            INTO RAM_AREA_1;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec         INTO ROM_AREA_2;
  codeSec, .text   INTO ROM_AREA_1;
END
INIT entry               /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* Initialization of the reset vector. */
```

Figure 6-5. Defining Multiple RAM and ROM Areas

According to the PRM file in **Figure 6-2**:

- The section `dataSec` will be allocated starting at `0x0900`.

- The section `constsec` will be allocated starting at `0x0C00`.

- The section `codeSec` will be allocated starting at `0x0B00`.

6.4.3 Relocatable versus Absolute Section

Generally, developing an application using relocatable sections is recommended. Relocatable sections offer several advantages.

6.4.3.1 Modularity

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

6.4.3.2 Multiple Developers

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

- An include file must be available for each assembly source file containing XREF directives for each exported variable, constant, and function. Additionally, the interface to the function should be described (parameter passing rules and 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. For instance, parameters are passed as expected and return value is retrieved correctly.

6.4.3.3 Early Development

The application can be developed before the application memory map is known. Often the definitive application memory map can be determined only once the size required for code and data can be evaluated. The size required for code or data can be quantified only 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.

6.4.3.4 Enhanced Portability

Since the memory map is not the same for all MCU derivatives, using relocatable sections allows the user to easily port the code to another MCU. When porting relocatable code to another target, link the application again with the appropriate memory map.

6.4.3.5 Tracking Overlaps

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

6.4.3.6 Reusability

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

For More Information On This Product,
Go to: www.freescale.com

Section 7. Assembler Syntax

7.1 Contents

7.2 Introduction

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

- Source line

7.3 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 text. 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.

Example:

    ;   This is a comment line

7.4 Source Line

Each source statement includes one or more of four fields:

1. A label

2. An operation field

3. One or several operands

4. A comment

Characters on the source line are case insensitive. Directives and instructions are also case insensitive. Symbols are case sensitive except when CI (option specifying case insensitivity for label names) is activated.

7.4.1 Label Field

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

NOTE: *For compatibility with other macro assembler vendors, an identifier starting on column 1 is considered to be a label, even if it is not terminated by a colon. When option* `-MCUasm` *(switch on MCUasm compatibility) is activated, labels must be terminated with a colon. An error message is issued, if 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 underline symbol (_). 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`.

7.4.2 Operation Field

The operation field follows the label field and is separated by 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 mnemonic
- A directive name
- A macro name

7.4.2.1 Instructions

Executable instructions for the M68HC12 processor are defined in the *CPU Reference Manual*, Motorola document order number CPU12RM/AD.

Table 7-1 presents a summary of available executable instructions.

Table 7-1. Executable Instructions (Sheet 1 of 8)

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
ANDA	Logical AND with accumulator A
ANDB	Logical AND with accumulator B
ANDCC	Logical AND with CCR
ASL	Arithmetic shift left in memory
ASLA	Arithmetic shift left accumulator A
ASLB	Arithmetic shift left accumulator B
ASLD	Arithmetic shift left accumulator D
ASR	Arithmetic shift left in memory
ASRA	Arithmetic shift right accumulator A
ASRB	Arithmetic shift right accumulator B
BCC	Branch if carry clear
BCLR	Clear bits in memory
BCS	Branch if carry set (same as BLO)
BEQ	Branch if equal
BGE	Branch if greater than or equal

Table 7-1. Executable Instructions (Sheet 2 of 8)

Instruction	Description
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
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 is clear
BRN	Branch never
BRSET	Branch if bits are set
BSET	Set bits in memory
BSR	Branch subroutine
BVC	Branch if overflow is cleared
BVS	Branch if overflow is set
CALL	Call subroutine in extended memory
CBA	Compare accumulator A and B
CLC	Clear carry bit
CLI	Clear interrupt bit
CLR	Clear memory
CLRA	Clear accumulator A
CLRB	Clear accumulator B

Table 7-1. Executable Instructions (Sheet 3 of 8)

Instruction	Description
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
CPD	Compare accumulator D and memory
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 null
DBNE	Decrement counter and branch if not null
DEC	Decrement memory location
DECA	Decrement accumulator A
DECB	Decrement accumulator B
DES	Decrement register SP
DEX	Decrement index register X
DEY	Decrement index register Y
EDIV	Unsigned division 32-bits/16 bits
EDIVS	Signed division 32-bits/16 bits
EMACS	Multiply and accumulate signed
EMAXD	Get maximum of 2 unsigned integers in accumulator D
EMAXM	Get maximum of 2 unsigned integers in memory
EMIND	Get minimum of 2 unsigned integers in accumulator D
EMINM	Get minimum of 2 unsigned integers in memory
EMUL	16-bit * 16-bit multiplication (unsigned)
EMULS	16-bit * 16-bit multiplication (signed)

Freescale Semiconductor, Inc.

Table 7-1. Executable Instructions (Sheet 4 of 8)

Instruction	Description
EORA	Logical XOR with accumulator A
EORB	Logical XOR with accumulator B
ETBL	16-bit table lookup and interpolate
EXG	Exchange register content
FDIV	16-bit / 16-bit fractional divide
IBEQ	Increment counter and branch if null
IBNE	Increment counter and branch if not null
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
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 is clear
LBCS	Long branch if carry is set
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

Table 7-1. Executable Instructions (Sheet 5 of 8)

Instruction	Description
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 is clear
LBVS	Long branch if overflow is set
LDAA	Load accumulator A
LDAB	Load accumulator B
LDD	Load accumulator D
LDS	Load register SP
LDX	Load index register X
LDY	Load index register Y
LEAS	Load SP with effective address
LEAX	Load X with effective address
LEAY	Load Y with effective address
LSL	Logical shift left in memory
LSLA	Logical shift left accumulator A
LSLB	Logical shift left accumulator B
LSLD	Logical shift left accumulator D
LSR	Logical shift left in memory
LSRA	Logical shift right accumulator A
LSRB	Logical shift right accumulator B
LSRD	Logical shift right accumulator D
MAXA	Get maximum of 2 unsigned bytes in accumulator A
MAXM	Get maximum of 2 unsigned bytes in memory
MEM	Membership function

For More Information On This Product,
Go to: www.freescale.com

Table 7-1. Executable Instructions (Sheet 6 of 8)

Instruction	Description
MINA	Get minimum of 2 unsigned bytes in accumulator A
MINM	Get minimum of 2 unsigned bytes in memory
MOVB	Memory to memory byte move
MOVW	Memory to memory word move
MUL	8 * 8 bit unsigned multiplication
NEG	2's complement in memory
NEGA	2's complement accumulator A
NEGB	2's complement accumulator B
NOP	No operation
ORAA	Logical OR with accumulator A
ORAB	Logical OR with accumulator B
ORCC	Logical OR with CCR
PSHA	Push register A
PSHB	Push register B
PSHC	Push register CCR
PSHD	Push register D
PSHX	Push register X
PSHY	Push register Y
PULA	Pop register A
PULB	Pop register B
PULC	Pop register CCR
PULD	Pop register D
PULX	Pop register X
PULY	Pop register Y
REV	MIN-MAX rule evaluation for 8-bit values
REVW	MIN-MAX rule evaluation for 16-bit values
ROL	Rotate memory left
ROLA	Rotate accumulator A left

Table 7-1. Executable Instructions (Sheet 7 of 8)

Instruction	Description
ROLB	Rotate accumulator B left
ROR	Rotate memory right
RORA	Rotate accumulator A right
RORB	Rotate accumulator B 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
SEC	Set carry bit
SEI	Set interrupt bit
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
SWI	Software interrupt
TAB	Transfer A to B
TAP	Transfer A to CCR
TBA	Transfer B to A

Table 7-1. Executable Instructions (Sheet 8 of 8)

Instruction	Description
TBEQ	Test counter and branch if null
TBL	8-bit table lookup and interpolate
TBNE	Test counter and branch if not null
TFR	Transfer register to register
TPA	Transfer CCR to A
TRAP	Software interrupt
TST	Test memory for 0 or minus
TSTA	Test accumulator A for 0 or minus
TSTB	Test accumulator B for 0 or minus
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

7.4.2.2 Directives

Assembler directives are described in **Section 8. Assembler Directives**.

7.4.2.3 Macro Name

A user-defined macro can be invoked in the assembler source program. This results in expansion of the code defined in the macro. Defining and using macros are described in **Section 9. Macros**.

7.4.3 Operand Fields

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

The address mode notations in **Table 7-2** are allowed in the operand field.

Table 7-2. Addressing Mode Notations

Addressing Mode	Notation
Inherent	No operands
Direct	<8-bit address>
Extended	<16-bit address>
Relative	<PC relative, 8-bit offset> or <PC relative, 16-bit offset>
Immediate	#<immediate 8-bit expression> or #<immediate 16-bit expression>
Indexed, 5-bit offset	<5-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, 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-Indirect, D accumulator offset	[D, xysp]

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

7.4.3.1 Inherent

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

Example:

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

7.4.3.2 Immediate

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

Example:

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

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

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

Example:

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

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

One very common programming error is to omit the # character. This causes the assembler to misinterpret the expression as an address rather than explicit data.

Example:

```
LDAA   $60
```

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

7.4.3.3 Direct

The direct addressing mode is used to access operands in the direct page of the memory (location $0000 to $00FF).

Access to this memory range (also called zero page) is faster and requires less code than the extended addressing mode (see next example). To speed up the application, a programmer can place the most commonly accessed data in this area of memory.

Example:

```
            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 register A is stored in the variable data which is located at address $50.

7.4.3.4 Extended

The extended addressing mode is used to access memory located above the direct page in a 64-Kbyte memory map.

Example:

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

In this example, the value in register A is stored in the variable `data`. This variable is located at address $0100 in the memory map.

7.4.3.5 Relative

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

Short branch instructions (BRA, BEQ, etc.) expect a signed offset encoded on one byte. The valid range for a short branch offset is [–128...127].

Example:

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

In this example, after the two `NOP`s have been executed, the application branches on the first `NOP` and continues execution.

Long branch instructions (LBRA, LBEQ, etc.) expect a signed offset encoded on two bytes. The valid range for a long branch offset is [–32,768...32,767].

Using the special symbol for location counter, it is possible also to specify an offset to the location pointer as the target for a branch instruction. The * (asterisk) refers to the beginning of the instruction where it is specified.

Example:

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

In this example, after the two NOPs have been executed, the application branches at offset –2 from the BRA instruction (for instance, on label main).

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

- A label defined in any absolute section

- A label defined in any relocatable section

- An external label, defined in an XREF directive

- An absolute EQU or SET label

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

- A label defined in any absolute section

- A label defined in any relocatable section

- An external label, defined in an XREF directive

7.4.3.6 Indexed, 5-Bit Offset

This addressing mode adds a 5-bit signed offset to the base index register to form the memory address, which 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 indexed PC and indexed PC relative addressing modes, see **7.4.3.16 Indexed PC versus Indexed PC Relative Addressing Mode**.

This addressing mode may be used to access elements in an n-element table, whose size is smaller than 16 bytes.

Example:

```
                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
```

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).

7.4.3.7 Indexed, 9-Bit Offset

This addressing mode adds 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 **7.4.3.16 Indexed PC versus 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.

Example:

```
          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).

7.4.3.8 Indexed, 16-Bit Offset

This addressing mode adds 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 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 modes, see **7.4.3.16 Indexed PC versus Indexed PC Relative Addressing Mode**.

Example:

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

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

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).

7.4.3.9 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 modes, see **7.4.3.16 Indexed PC versus Indexed PC Relative Addressing Mode**.

Example:

```
                ORG $1000
    CST_TBL1:   DC.W  $1020, $1050, $2001
                ORG $2000
    CST_TBL2:   DC.B  $10, $35, $46
                ORG $3000
    main:
                LDX #$CST_TBL
                LDAA [4,X]
```

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. Accumulator A is loaded with $35 and the value is stored at address $2001.

7.4.3.10 Indexed, Pre-Decrement

This addressing mode allows the user 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.

Example:

```
                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
```

Base register X is loaded with the address of the element following the table `CST_TBL` ($1006).

Register X 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 register D.

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

7.4.3.12 Indexed, Post-Decrement

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

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

Example:

```
                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
```

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. Register X is decremented by 2 (its value is $1004).

Register 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 2 again (its value is now $1002).

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

7.4.3.13 Indexed, Post-Increment

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

Example:

```
                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
```

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 register X is incremented by 1 (its value is $1001).

Register X is not equal to the address of END_TBL, so the value at address $1001 ($10) is added to register D and then register X is incremented by 1 (its value is $1002).

This loop is repeated as long as register X did not reach the end of the table END_TBL ($1006).

7.4.3.14 Indexed, Accumulator Offset

This addressing mode adds 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.

Example:

```
main:
          LDAB #$20
          LDX  #$600
          LDAA B,X

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

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.

7.4.3.15 Indexed-Indirect, D Accumulator Offset

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

Example:

```
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
```

This example represents a jump table. The values beginning at goto1 are potential destinations 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 label entry2) and jumps to that location.

7.4.3.16 Indexed PC versus Indexed PC Relative Addressing Mode

When using the indexed addressing mode with PC as the base register, the macro assembler allows use of either indexed PC (<offset>, PC) or indexed PC relative (<offset>, PCR) notation.

When indexed PC notation is used, the offset specified is inserted directly in the opcode.

Example:
```
main:
                LDAB  3, PC
                DC.B  $20, $30, $40, $50
```

In the previous example, 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.

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

In the previous example, register B is loaded with the value stored at label x4 ($50). The macro assembler evaluates the offset between the current location counter and the symbol x4 to determine the value, which must be stored in the opcode.

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

- A label defined in any absolute section

- A label defined in any relocatable section

- An external label, defined in an XREF directive

- An absolute EQU or SET label

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

- A label defined in any absolute section

- A label defined in any relocatable section

- An external label, defined in an XREF directive

7.4.4 Comment Field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in the comment field.

Example:

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

7.5 Symbols

The following sections describe symbols used by the assembler.

7.5.1 User-Defined Symbols

Symbols identify memory locations in program or data sections in an assembly module.

A symbol has two attributes:

- The section in which the memory location is defined

- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION 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. In the next example, `labelx` is used to represent a symbol.

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Section
label2: DC.B 5 ; label2 is assigned offset 2 within Section
label3: DC.B 1 ; label3 is assigned offset 7 within Section
```

Figure 7-1. Relocatable Symbols Program Example

It is also possible to define a label with either an absolute or previously defined relocatable value, using a SET or EQU directive.

Symbols with absolute values must be defined with constant expressions.

```
 Sec: SECTION
label1: DC.B 2            ; label1 is assigned offset 0
                         ; within Section
label2: EQU 5            ; label2 is assigned value 5
label3: EQU label1       ; label3 is assigned address of
                         ; label1
```

Figure 7-2. Set or EQU Directive Program Example

7.5.2 External Symbols

A symbol can be made external using the XDEF directive. In another source file, an XREF or XREFB directive may reference the symbol. Since its address is unknown in the referencing file, it is considered to be relocatable.

```
 XREF extLabel      ; symbol defined in another 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
```

Figure 7-3. External Symbol Program Example

7.5.3 Undefined Symbols

If a label is neither defined in the source file nor declared external using XREF or XREFB, the assembler considers it to be undefined and generates an error.

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

Figure 7-4. Undefined Symbol Example

7.5.4 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
```

Additionally, the keyword PAGE is also a reserved identifier. It is used to refer to bits 16–23 of a 24-bit value.

7.6 Constants

The assembler supports integer and ASCII string constants.

7.6.1 Integer Constants

The assembler supports four representations of integer constants:

- A decimal constant is defined by a sequence of decimal digits (0–9). Example: 5, 512, 1024

- A hexadecimal constant is defined by a dollar character ($) followed by a sequence of hexadecimal digits (0–9, a–f, A–F). Example: $5, $200, $400

- An octal constant is defined by the 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 constants is decimal, but it can be changed using the BASE directive. When the default base is not decimal, decimal values cannot be represented because they do not have a prefix character.

7.6.2 String Constants

A string constant is a series of printable characters enclosed in single quote (') or double quotes ("). Double quotes are only allowed within strings delimited by single quotes. Single quotes are only allowed within strings delimited by double quotes.

Example**:**

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

7.6.3 Floating-Point Constants

The macro assembler does not support floating-point constants.

7.7 Operators

The following subsections describe the operators used in expressions that are recognized by the assembler.

7.7.1 Addition and Subtraction Operators (Binary)

Syntax: Addition: <operand> + <operand>

Subtraction: <operand> – <operand>

Description: The + (plus) operator adds two operands, whereas the – (minus) operator subtracts them. The operands can be any expression evaluating to an absolute or relocatable expression. Note that addition between two relocatable operands is not allowed.

Example:
```
$A3216 + $42; Addition of 2 absolute operands ( = $A3258)
label – $10 ; Subtraction with value of 'label'
```

7.7.2 Multiplication, Division, and Modulo Operators (Binary)

Syntax: Multiplication: <operand> * <operand>

Division: <operand> / <operand>

Modulo: <operand> % <operand>

Description: The * (asterisk) operator multiplies two operands, the / (slash) operator performs an integer division of the two operands and returns the quotient of the operation. The % (percent) 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 0.

Example:
```
23 * 4    ; multiplication ( = 92)
23 / 4    ; division ( = 5)
23 % 4    ; remainder( = 3)
```

7.7.3 Sign Operators (Unary)

Syntax: Plus: +<operand>

Minus: –<operand>

Description: The + (plus) operator does not change the operand, whereas the
– (minus) operator changes the operand to its two's
complement. These operators are only valid for absolute
expression operands.

Example:
```
+$32        ; ( = $32)
–$32        ; ( = $CE = –$32)
```

7.7.4 Shift Operators (Binary)

Syntax: Shift left: <operand> << <count>

Shift right: <operand> >> <count>

Description: The << (double less than) operator shifts the left operand left
by the number of bytes specified in right operand.

The >> (double greater than) operator shifts the left operand
right by the number of bytes specified in right operand.
Operands can be any expression evaluating to an absolute
expression.

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

7.7.5 Bitwise Operators (Binary)

Syntax: Bitwise AND: <operand> & <operand>

                   Bitwise OR: <operand> | <operand>

                   Bitwise XOR: <operand> ^ <operand>

Description: The & (ampersand) operator performs an AND between the two operands at the bit level.

The | (vertical bar) operator performs an OR between the two operands at the bit level.

The ^ (caret) operator performs an XOR between the two operands at the bit level.

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

Example:

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

7.7.6 Bitwise Operators (Unary)

Syntax: One's complement: ~<operand>

Description: The ~ (tilde) operator evaluates the one's complement of the operand.

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

Example:

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

7.7.7 Logical Operators (Unary)

Syntax: Logical NOT: !<operand>

Description: The ! (exclamation point) operator returns 1 (true) if the
operand is 0; otherwise, it returns 0 (false).

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

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

7.7.8 Relational Operators (Binary)

Syntax:

Equal: <operand>   = <operand>
                <operand>   == <operand>
Not equal: <operand>   != <operand>
                <operand>   <> <operand>
Less than: <operand>   < <operand>
Less than or equal: <operand> <= <operand>
Greater than: <operand> > <operand>
Greater than or equal:<operand> >= <operand>

Description: These operators compare the two operands and return 1 if the
condition is true or 0 if the condition is false.

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

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

7.7.9 Memory PAGE Operator (Unary)

Syntax: Get allocation page: PAGE(<operand>)

Description: The PAGE operator returns the page number where the operand is allocated. For a value coded on four bytes, the PAGE operator returns the content of bits 23 to 16 of the value.

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

When the PAGE operator is used with an absolute expression, the assembler evaluates the page directly and the value is written to the output file. In this case, PAGE(<operand>) is equivalent to <operand> & $FF0000.

Example:

```
#PAGE($D)          ; = 0
#PAGE($15A352)     ; = $15
#PAGE(label)       ; = Page number where label is
                   ; allocated
```

7.7.10 Force Operator (Unary)

Syntax: 8-bit address:<<operand>

<operand>.B

16-bit address:><operand>

<operand>.W

Description: The < (less than) or .B operators force the operand to be an 8-bit operand, whereas the > or .W operators force the operand to be a 16-bit operand.

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

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

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

Example:

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

Operator precedence follows the rules for ANSI C operators. See **Table 7-3**.

Table 7-3. Operator Precedence

Operator	Description	Associativity
()	Parenthesis	Right to left
< > .B .W PAGE	Force direct address, index, or immediate value to 8 bits. Force direct address, index, or immediate value to 16 bits. Force direct addressing mode for absolute address. Force extended addressing mode for absolute address. Access 4-bit page number (bits 16–23 of 20-bit value).	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

Table 7-3. Operator Precedence (Continued)

Operator	Description	Associativity
 <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to right
=, == !=, <>	Equal to Not Equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

7.8 Expressions

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 * (asterisk) represents the value of the location counter at the beginning of the instruction or directive, even if 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 to one of the following types 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.

7.8.1 Absolute Expressions

Expressions involving constants, known as absolute labels, or expressions are absolute expressions. An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

Example of 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:

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

This can be simplified as:

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

In the following example, 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`.

```
DataSec: SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1

CodeSec: SECTION
entry:
        LDD #tabEnd-tabBegin <- Absolute expression
```

7.8.2 Simple Relocatable Expression

A simple relocatable expression results from an operation such as the one shown here:

- <relocatable expression> + <absolute expression>

- <relocatable expression> – <absolute expression>

- <absolute expression> + < relocatable expression>

Example:

```
          XREF XtrnLabel
DataSec: SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1

CodeSec: SECTION
entry:
   LDA tabBegin+2  <- Simple relocatable expression
   BRA *-3         <- Simple relocatable expression
   LDA XtrnLabel+6 <- Simple relocatable expression
```

Table 7-4 indicates the type of expression according to the operator in an unary operation.

Table 7-4. Expression — Operator Relationship (Unary)

Operator	Operand	Expression
—, !, ~	Absolute	Absolute
—, !, ~	Relocatable	Complex
+	Absolute	Absolute
+	Relocatable	Relocatable

Table 7-5 describes the type of expression according to left and right operands in a binary operation.

Table 7-5. Expression — Operator Relationship (Binary)

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

7.9 Translation Limits

These limitations apply to the macro assembler:

- Floating-point constants are not supported.

- Complex relocatable expressions are not supported.

- Lists of operands or symbols must be separated with a comma.

- Include may be nested up to 50.

- The maximum line length is 1023.

Section 8. Assembler Directives

8.1 Contents

For More Information On This Product,
Go to: www.freescale.com

For More Information On This Product,
Go to: www.freescale.com

8.2 Introduction

This chapter introduces assembler directives. Functional descriptions and examples for each directive are provided. The following tables give an overview of the different directives.

8.2.1 Section Definition Directives

The directives in **Table 8-1** define new sections.

Table 8-1. Section Directives

Directive	Description
ORG	Defines an absolute section
SECTION	Defines a relocatable section
OFFSET	Defines an offset section

8.2.2 Constant Definition Directives

The directives in **Table 8-2** define assembly constants.

Table 8-2. Constant Directives

Directive	Description
EQU	Assigns a name to an expression, cannot be redefined
SET	Assigns a name to an expression, can be redefined

8.2.3 Data Allocation Directives

The directives in **Table 8-3** allocate variables.

Table 8-3. Data Allocation Directives

Directive	Description
DC	Defines a constant variable
DCB	Defines a constant block
DS	Defines storage for a variable

MCUez HC12 Assembler

User's Manual

8.2.4 Symbol Linkage Directives

The directives in **Table 8-4** export or import global symbols.

Table 8-4. Symbol Linkage Directives

Directive	Description
ABSENTRY	Specifies the application entry point when an absolute file is generated
XDEF	Makes a symbol public, visible from outside
XREF	Imports reference to an external symbol
XREFB	Imports reference to an external symbol located on the direct page

8.2.5 Assembly Control Directives

The general-purpose directives in **Table 8-5** control the assembly process.

Table 8-5. Assembly Control Directives

Directive	Description
ALIGN	Defines alignment constraint
BASE	Specifies default base for constant definition
END	End of assembly unit
EVEN	Defines 2-byte alignment constraint
FAIL	Generates user-defined error or warning messages
INCLUDE	Includes text from another file
LONGEVEN	Defines 4-byte alignment constraint

8.2.6 Listing File Control Directives

The directives in **Table 8-6** control generation of the assembler listing file.

Table 8-6. Assembler List File Directives

Directive	Description
CLIST	Specifies if all instructions in a conditional assembly block must be inserted in the listing file or not
LIST	Specifies that all subsequent instructions must be inserted in the listing file
LLEN	Defines line length in assembly listing file
MLIST	Specifies if macro expansions must be inserted in the listing file
NOLIST	All subsequent instructions will not be inserted in the listing file
NOPAGE	Disables paging in the assembly listing file
PAGE	Inserts page break
PLEN	Defines page length in the assembler listing file
SPC	Inserts an empty line in the assembly listing file
TABS	Defines number of characters to insert in the assembler listing file for a TAB character
TITLE	Defines the user-defined title for the assembler listing file

8.2.7 Macro Control Directives

The directives in **Table 8-7** are used for the definition and expansion of macros.

Table 8-7. Macro Directives

Directive	Description
ENDM	End of user-defined macro
MACRO	Start of user-defined macro
MEXIT	Exit from macro expansion

8.2.8 Conditional Assembly Directives

The directives in **Table 8-8** are used for conditional assembling.

Table 8-8. Conditional Assembly Directives

Directive	Description
ELSE	Alternate of conditional block
ENDIF	End of conditional block
IF	Start of conditional block. A Boolean expression follows this directive.
IFC	Tests if two string expressions are equal
IFDEF	Tests if a symbol is defined
IFEQ	Tests if an expression is null
IFGE	Tests if an expression is greater than or equal to 0
IFGT	Tests if an expression is greater than 0
IFLE	Tests if an expression is less than or equal to 0
IFLT	Tests if an expression is less than 0
IFNC	Tests if two string expressions are different
IFNDEF	Tests if a symbol is undefined
IFNE	Tests if an expression is not null

8.3 ABSENTRY — Application Entry Point

Syntax: ABSENTRY <label>

Description: This directive specifies the application entry point in a directly generated absolute file (the option -FA2 *ELF/DWARF 2.0* absolute file must be enabled).

Using this directive, the entry point is written in the ELF 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.

Example: If the next example is assembled using the -FA2 option, an *ELF/DWARF 2.0* absolute file is generated.

```
 ABSENTRY entry


        ORG $fffe
 Reset: DC.W entry


        ORG $70
 entry: NOP
        NOP
 main:
        NOP
        BRA main
```

8.4 ALIGN — Align Location Counter

Syntax: ALIGN <n>

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 32,767. The ALIGN directive can force alignment to any size. The filling bytes inserted for alignment purposes are initialized with \0.

ALIGN can be used in code or data sections.

Example: The following example aligns the HEX label to a location, which is a multiple of 16 (in this case, location 00010 hex).

```
000000 4849 4748    DC.B  "HIGH"
000004 0000 0000    ALIGN 16
000008 0000 0000
00000C 0000 0000
000010 007F         HEX: DC.W 127    ; HEX is allocated on
                                     ; an address which is
                                     ; a multiple of 16.
```

8.5 BASE — Set Number Base

Syntax: BASE <n>

Description: This 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, and 16. Unless a
default base is specified using the BASE directive, the default
number base is decimal.

Example:

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

NOTE: *Even if the base value is set to 16, hexadecimal constants terminated by a D
must be prefixed by the $ (dollar sign) character; otherwise, they are
interpreted as decimal constants in old style format. For example, constant 45D
is interpreted as decimal constant 45, not as hexadecimal constant $45D.*

8.6 CLIST — List Conditional Assembly

Syntax: CLIST [ON | OFF]

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 ON is specified with the CLIST directive, the listing file includes all directives and instructions in the conditional assembly block, even those that do not generate code.

When OFF is specified, only the directives and instructions that generate code are listed.

When the option -L is activated, the assembler defaults to CLIST ON.

Example: Listing file with CLIST OFF:

```
        CLIST OFF
Try:EQU       0
        IFEQ     Try
          LDD    #1023
        ELSE
          LDD    #0
        ENDIF
```

The corresponding listing file is:

```
Try:EQU 0
                        IFEQ    Try
37 B5 03 FF              LDD  #1023
                        ELSE
                        ENDIF
```

Example: Listing file with `CLIST ON`:

When assembling the code:

```
        CLIST ON
Try:EQU       0
        IFEQ      Try
          LDD     #1023
        ELSE
          LDD     #0
        ENDIF
```

The corresponding listing file is:

```
             Try:EQU 0
                 IFEQ    Try
37 B5 03 FF          LDD   #1023
                 ELSE
                   ADD   #0
                 ENDIF
```

8.7 DC — Define Constant

Syntax: [<label>:] DC [<size>] <expression> [, <expression>]...

where

<size> = B (default), W, or L.

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). Alternately, 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.

These 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.

- DC.W — Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a 2-byte boundary.

- DC.L — Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a 4-byte boundary.

Example for DC.B:
```
000000 4142 4344    Label: DC.B "ABCDE"
000004 45
000005 0A0A 010A           DC.B %1010, @12, 1, $A
000009 xx                  DC.B PAGE(Label)
```

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
```

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.

8.8 DCB — Define Constant Block

Syntax:

[<label>:] `DCB` [<size>] <count>, <value>

where

<size> = B (default), W, or L

Description:

The `DCB` directive causes the assembler to allocate a memory block initialized with the specified <value>. The length of the block is <size> * <count>.

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

The value of each storage unit allocated is the sign-extended expression <value>, which may contain forward references. The <count> cannot be relocatable. This directive does not perform alignment.

These rules apply to size specifications for `DCB` directives:

- `DCB.B` — One byte is allocated for numeric expressions.

- `DCB.W` — Two bytes are allocated for numeric expressions.

- `DCB.L` — Four bytes are allocated for numeric expressions.

Example:

```
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
```

8.9 DS — Define Space

Syntax: [<label>:] DS [.<size>] <count>

where

<size> = B (default), W, or L

Description:

The DS directive is used to reserve memory for variables. The content of the reserved memory is not initialized. The length of the block is <size> * <count>.

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

Example:

```
Counter: DS.B  2   ; 2 contiguous bytes in memory
         DS.B  2   ; 2 contiguous bytes in memory
                   ; can only be accessed through the
                   ; label Counter
         DS.L  5   ; 5 contiguous longwords in memory
```

The label, Counter, references the lowest address of the defined storage area.

For More Information On This Product,
Go to: www.freescale.com

8.10 ELSE — Conditional Assembly

Syntax: IF <condition>

  [<assembly language statements>]

[ELSE]

  [<assembly language statements>]

ENDIF

Description: If <condition> is true, the statements between IF and the corresponding ELSE directive generate code.

If <condition> is false, the statements between ELSE and the corresponding ENDIF directive generate code. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Example: This is an example of using conditional assembly directives:

```
Try: EQU 0
     IF  Try != 0
       LDD   #1023
     ELSE
       LDD   #0
     ENDIF
```

The value of Try determines the instruction that generates code. As shown, the LDD #1023 instruction generates code. Changing the operand of the equ directive to 1 causes the LDD #0 instruction to generate code instead.

The following shows the listing provided by the assembler for these lines of code:

```
         0000 0001    Try:  EQU   0
         0000 0001          IF  Try != 0
                                ELSE
000000 CC 0000                 LDD #0
                             ENDIF
```

8.11 END — End Assembly

Syntax: `END`

Description: The `END` directive indicates the end of the source code.
               Subsequent source statements in this file are ignored. An `END`
               directive in included files causes subsequent source statements
               in the include file to be skipped.

Example: When assembling the code:

```
Label: NOP
       NOP
       NOP
       END

       NOP ; No code generated
       NOP ; No code generated
```

The generated listing file is:

```
000000 A7     Label: NOP
000001 A7            NOP
000002 A7            NOP
                     END
```

8.12 ENDIF — End Conditional Assembly

Syntax: ENDIF

Description: The ENDIF directive indicates the end of a conditional block.
               Nesting of conditional blocks is allowed. The maximum level
               of nesting is limited by the available memory at assembly time.

Example: See an example of directive IF in **8.17 IF — Conditional
               Assembly**.

8.13 ENDM — End Macro Definition

Syntax: ENDM

Description: The ENDM directive terminates both the macro definition and
               macro expansion.

Example:

```
5    5              cpChar: MACRO   ; start macro definition
6    6                              LDD \1
7    7                              STD \2
8    8               ENDM           ; end of macro definition
9    9              codeSec: SECTION
10   10             Start:
11   11                             cpChar char1, char2
12    6m  000000 FC xxxx   +        LDD char1
13    7m  000003 7C xxxx   +        STD char2
14   12   000006 A7                 NOP
15   13   000007 A7                 NOP
```

8.14 EQU — Equate Symbol Value

Syntax: <label>: EQU <expression>

Description: The EQU directive assigns the value of the <expression> in the operand field to <label>. The <label> and <expression> fields are both required, and the <label> cannot be defined anywhere else in the program. The <expression> cannot include a symbol that is undefined or not yet defined.

                   The EQU directive does not allow forward references.

Example:

```
        0000 0014  MaxElement: EQU  20
        0000 0050  MaxSize:    EQU  MaxElement * 4

  000000            Time:   DS.W 3
        0000 0000  Hour:   EQU Time  ; first word addr
        0000 0002  Minute: EQU Time+2; second word addr
        0000 0004  Second: EQU Time+4; third word addr
```

8.15 EVEN — Force Word Alignment

Syntax: EVEN

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
longword operations to begin at even address boundaries. In
such cases, the use of the EVEN directive ensures correct
alignment. Omission of the directive can result in an error
message.

Example:
```
6    6   000000                    ds.w  2
     ; location count has an even value, no padding
     ; inserted.
7    7                             even
8    8   000004                    ds.b  1
     ; location count has an odd value, one padding byte
     ; inserted.
9    9   000005 00                 even
10   10 000006                     ds.b  3
     ; location count has an odd value, one padding byte
     ; inserted.
12   12         0000 000A aaa:  equ   10
```

8.16 FAIL — Generate Error Message

Syntax: FAIL <arg> | <string>

Description: FAIL directive operation depends on the operand specified. If
               <arg> is in the range [500–$FFFFFFFF], the assembler
               generates a warning message, including the line number and
               argument of the directive.

Example: The following code segment:

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

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

Generates this error message:

```
>> in "Y:\DEMO\HC12A\cbe.asm", line 14, col 19, pos 242

        IFC "\2", ""
          FAIL 600
                ^
WARNING A2332: FAIL found
```

If <arg> is 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.

The following code segment:

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

          IFC "\2", ""
            FAIL 600
          ELSE
            STD \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
          cpChar , char2
```

Generates this error message:

```
>> in "Y:\DEMO\HC12A\cbe.asm",line 7,col 19, pos 112

        IFC "\1", ""
          FAIL 200
                ^
ERROR A2329: FAIL found
```

If a string is supplied as the operand, the assembler generates an error message, including the line number and <string>. The assembler does not generate an object file.

Example: The following code segment:

```
cpChar: MACRO
        IFC "\1", ""
          FAIL "A character must be
specified as first parameter"
          MEXIT
        ELSE
          LDD \1
        ENDIF

        IFC "\2", ""
          FAIL 600
        ELSE
          STD \2
        ENDIF
      ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

Generates this error message:

```
>> in "Y:\DEMO\HC12A\cbe.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
```

The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

8.17 IF — Conditional Assembly

Syntax: IF <condition>

                     [<assembly language statements>]

                     [ELSE]

                     [<assembly language statements>]

                     ENDIF

Description: If <condition> is true, the statements immediately following
the IF directive generate code. Assembly continues until the
corresponding ELSE or ENDIF directive is reached. Then all
statements until the corresponding ENDIF directive are
ignored. Nesting of conditional blocks is allowed. The
maximum level of nesting is limited by available memory at
assembly time.

The expected syntax for <condition> is:

```
<condition> := <expression> <relation> <expression>
<relation> :=  "=" | "!=" | " >=" | ">" | "<=" | "<" | "<>"
```

The <expression> must be absolute. It must be known at
assembly time.

Example:

This is an example of using conditional assembly directives:

```
Try: EQU 0
     IF  Try != 0
       LDD   #1023
     ELSE
       LDD   #0
     ENDIF
```

The value of TRY determines the instruction that generates
code. As shown, the LDD #0 instruction generates code.
Changing the operand of the EQU directive to one causes the
LDD #1023 instruction to generate code instead.

This is the listing provided by the assembler for these lines of code:

```
0000 0000    Try:    EQU    0
0000 0000            IF  Try != 0
                     ELSE
000000 CC 0000        LDD #0
                     ENDIF
```

8.18 IFCC — Conditional Assembly

Syntax: IFCC <condition>

                 [<assembly language statements>]

                 [ELSE]

                 [<assembly language statements>]

                 ENDIF

Description: These directives can be replaced by the IF directive. If 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 Types

IFCC	Condition	Meaning
IFEQ	<expression>	IF <expression> == 0
IFNE	<expression>	IF <expression> != 0
IFLT	<expression>	IF <expression> < 0
IFLE	<expression>	IF <expression> <= 0
IFGT	<expression>	IF <expression> > 0
IFGE	<expression>	IF <expression> >= 0
IFC	<string1>, <string2>	IF <string1> == <string2>
IFNC	<string1>, <string2>	IF <string1> != <string2>
IFDEF	<label>	IF <label> was defined
IFNDEF	<label>	IF <label> was not defined

Example: The following is an example of using conditional assembly directives:

```
Try: EQU 0
     IFNE  Try
       LDD #1023
      ELSE
       LDD #0
      ENDIF
```

The value of TRY determines the instruction to be assembled in the program. As shown, the LDD #0 instruction generates code. Changing the directive to IFEQ causes the LDD #1023 instruction to generate code instead.

The following shows the listing provided by the assembler for these lines of code:

```
              Try: EQU  0
                   IFNE Try
                   ELSE
00000 37 B5 00 00    LDD #0
                   ENDIF
```

8.19 INCLUDE — Include Text from Another File

Syntax: `INCLUDE <filename>`

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 <filename> relative to the current working directory. If the file is not found, then it is searched for in each path specified in the environment variable GENPATH.

Example: `INCLUDE "..\LIBRARY\macros.inc"`

Freescale Semiconductor, Inc.

8.20 LIST — Enable Listing

Syntax: LIST

Description: Specifies that the following instructions must be inserted in the listing and debug files. The listing file is only generated if the option -L is specified on the command line.

The source text following the LIST directive is listed until a NOLIST or an END is reached.

This directive is not written to the listing and debug file. When neither the LIST nor NOLIST directives are specified in a source file, all instructions are written to the list file.

Example: This portion of code:

```
aaa:    nop

        list
bbb:    nop
        nop

        nolist
ccc:    nop
        nop

        list

ddd:    nop
        nop
```

Generates this listing file:

```
1    1   000000 A7        aaa:    nop
2    2
4    4   000001 A7        bbb:    nop
5    5   000002 A7                nop
6    6
    13   13   000005 A7           ddd:    nop
14   14   000006 A7                 nop
```

The gap in the location counter is due to instructions defined in the NOLIST-LIST block.

See also: **8.26 NOLIST — Disable Listing**

8.21 LLEN — Set Line Length

Syntax: LLEN <n>

Description: Sets the number of characters, <n>, from the source line that are included on the listing line. The values allowed for <n> are in the range [0 – 132]. If a value smaller than 0 is specified, the line length is set to 0. If a value bigger than 132 is specified, the line length is set to 132.

Lines of the source file that exceed the specified number of characters are truncated in the listing file.

Example: This portion of code:

```
dc.b      5
llen      $20
dc.w      $4567, $2345
llen      $17
dc.w      $4567, $2345
even
nop
```

Generates this listing file:

```
Motorola HC12-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997


Abs. Rel.   Loc.   Obj. code    Source line
---- ----   ------ ---------    -----------
1    1      000000 05              dc.b      5
3    3
4    4      000001 4567 2345       dc.w      $4567, $2345
5    5
7    7      000005 4567 2345       dc.w      $4567
8    8      000009 00              even
9    9      00000A 9D              nop
10   10
```

The LLEN $17 directive causes the second dc.w $4567, $2345 to be truncated in the assembler listing file. The generated code is correct.

8.22 LONGEVEN — Forcing Longword Alignment

Syntax: `LONGEVEN`

Description: This directive forces the next instruction to the next long-word address relative to the start of the section. `LONGEVEN` is an abbreviation for `ALIGN 4`.

Example:

```
2    2    000000 01                   dcb.b 1,1
          ; location counter is not a multiple of 4, 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 A7                   nop
          ; location counter is not a multiple of 4, 3 filling bytes
          ; are required.
10   10   000001 0000 00              longeven
11   11   000004 A7                   nop
```

8.23 MACRO — Begin Macro Definition

Syntax: <label>: MACRO

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, refer to **Section 9. Macros**.

Example:
```
 5    5                  cpChar: MACRO; start macro definition
 6    6                          LDD \1
 7    7                          STD \2
 8    8                          ENDM ; end of macro definition
 9    9                  codeSec: SECTION
10   10                  Start:
11   11                  cpChar char1, char2
12    6m   000000 FC xxxx   +              LDD char1
13    7m   000003 7C xxxx   +              STD char2
14   12    000006 A7                  NOP
15   13    000007 A7                  NOP
```

8.24 MEXIT — Terminate Macro Expansion

Syntax: MEXIT

Description: MEXIT is usually used together with conditional assembly
                 within a macro. In that case, macro expansion might terminate
                 prior to terminating the macro definition. The MEXIT directive
                 causes macro expansion to skip any remaining source lines
                 ahead of the ENDM directive.

Example: This portion of code:

```
sav2: MACRO                 ; Start macro definition
         ldx  savet
         ldaa \1
         staa 0,x           ; save first argument
         ldaa \2
         staa 2,x           ; save second argument
         ifc  '\3', ''      ; is there a 3rd argument?
           mexit            ; no, exit from macro.
         endif
         ldaa \3            ; save third argument
         staa 4, x
        endm                ; End of macro definition
entry:
        sav2 char1, char2
```

Generates this listing file:

```
27  11m   000000 FE xxxx   +   ldx  savet
28  12m   000003 B6 xxxx   +   ldaa char1
29  13m   000006 6A00      +   staa 0,x;save first
                                             ; argument
30  14m   000008 B6 xxxx   +   ldaa char2
31  15m   00000B 6A02      +   staa 2,x ;save second
                                             ; argument
32  16m                    +
33  17m   0000 0001        + ifc  '','' ;is there a
                                           ; 3rd arg.
35  18m                    +   mexit    ;no,exit macro.
36  19m                    +   endif
37  20m                    +
38  21m                    +   ldaa     ;save third
                                           ; argument
39  22m                    +   staa 4, x
```

8.25 MLIST — List Macro Expansions

Syntax: MLIST [ON | OFF]

Description: When the ON keyword is entered with an MLIST directive, the
               assembler includes the macro expansions in the listing and
               debug files. When the OFF keyword is entered, macro
               expansions are omitted from the listing and debug files. This
               directive is not written to the listing and debug files, and the
               default value is ON.

Example: This listing shows a macro definition and expansion with
               MLIST ON:

```
10   10                      swap: macro
12   12                            ldx   \2
13   13                            std   \2
14   14                            stx   \1
15   15                               endm
16   16
18   18                            swap first, second
19   11m  000000 FC xxxx     +         ldd   first
20   12m  000003 FE xxxx     +         ldx   second
21   13m  000006 7C xxxx     +         std   second
22   14m  000009 7E xxxx     +         stx   first
23   19   00000C A7                NOP
```

For the same code, with MLIST OFF, the listing file is:

```
10   10                      swap: macro
11   11                            ldd   \1
12   12                            ldx   \2
13   13                            std   \2
14   14                            stx   \1
15   15                               endm
16   16
18   18                            swap first, second
23   19   00000C A7                NOP
```

8.26 NOLIST — Disable Listing

Syntax: `NOLIST`

Description: Suppresses printing of the following instructions in the assembly listing and debug files until a `LIST` directive is reached.

Example: This portion of code:

```
aaa:    nop

        list
bbb:    nop
        nop

        nolist
ccc:    nop
        nop

        list

ddd:    nop
        nop
```

Generates this listing file:

```
1    1   000000 A7   aaa:    nop
2    2
4    4   000001 A7   bbb:    nop
5    5   000002 A7           nop
6    6
     ; The gap in the location counter is due to
     ; instructions
     ; which are defined inside a NOLIST block.
13   13  000005 A7   ddd:    nop
14   14  000006 A7           nop
```

The gap in the location counter is due to instructions defined inside a `NOLIST` block.

8.27 NOPAGE — Disable Paging

Syntax: `NOPAGE`

Description: Disables pagination in the listing file. Program lines are listed continuously without headings or top or bottom margins.

8.28 ORG — Set Location Counter

Syntax: ORG <expression>

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.

Example:

```
        org   $2000
b1:     nop
b2:     rts
```

8.29 OFFSET — Create Absolute Symbols

Syntax: OFFSET <expression>

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.

The OFFSET section is useful to simulate data structure or a stack frame.

Example: The following example shows how OFFSET can be used to access elements of a structure.

```
 6   6                                   OFFSET 0
 7   7  000000       ID:      DS.B    1
 8   8  000001       COUNT:   DS.W    1
 9   9  000003       VALUE:   DS.L    1
10  10  0000 0007    SIZE:    EQU *
11  11
12  12               DataSec:SECTION
13  13  000000       Struct: DS.B SIZE
14  14
15  15               CodeSec:SECTION
16  16               entry:
17  17  000003 CE xxxx       LDX  #Struct
18  18  000006 8600          LDAA #0
19  19  000008 6A00          STAA ID, X
20  20  00000A 6201          INC COUNT, X
21  21  00000C 42            INCA
22  22  00000D 6A03          STAA VALUE, X
```

As soon as a statement affecting the location counter (other than EVEN, LONGEVEN, ALIGN, or DS) is encountered after the OFFSET directive, the offset section is ended. The preceding section is activated again, and the location counter is restored to the next available location in this section.

Example:

```
7    7                     ConstSec: SECTION
8    8   000000 11         cst1:      DC.B  $11
9    9   000001 11         cst2:      DC.B  $13
10   10
11   11                               OFFSET 0
12   12   000000           ID:        DS.B   1
13   13   000001           COUNT:     DS.W   1
14   14   000003           VALUE:     DS.L   1
15   15   0000 0007        SIZE:      EQU *
16   16
17   17   000002 22        cst3:      DC.B  $22
```

In the previous example, the symbol cst3, defined after the OFFSET directive, defines a constant byte value. This symbol is appended to the section ConstSec, which precedes the OFFSET directive.

8.30 PAGE — Insert Page Break

Syntax: PAGE

Description: Inserts a page break in the assembly listing

Example: The following portion of code:

```
codeSec: SECTION
            nop
            nop
            page
            nop
            nop
```

Generates this listing file:

```
Motorola HC12-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997


Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
1    1                         codeSec: SECTION
3    3      000000 A7                  nop
4    4      000001 A7                  nop


Motorola HC12-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997


Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
6    6      000002 A7                  nop
7    7      000003 A7                  nop
```

8.31 PLEN — Set Page Length

Syntax: PLEN <n>

Description: Sets the page length to <n> lines. <n> may range from 10 to
10,000. 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.

8.32 SECTION — Declare Relocatable Section

Syntax: <name>: SECTION [SHORT][<number>]

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 0. 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, where the same name is specified, refer to the same
section.

<number> is optional and only specified for compatibility with
the MASM assembler.

A section is a code section if it contains at least an 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 if it contains at least a DS directive or if it is
empty.

Example: The following example demonstrates the definition of a section
             `aaa`, which is split into two blocks, with section `bbb` between
             them. The location counter associated with label `zz` is 1,
             because a `NOP` instruction was already defined in this section
             at label `xx`.

```
2    2                              aaa:    section 4
3    3    000000 A7                 xx:      nop
4    4                              bbb:    section 5
5    5    000000 A7                 yy:     nop
6    6    000001 A7                         nop
7    7    000002 A7                         nop
8    8                              aaa:    section 4
9    9    000001 A7                 zz:      nop
```

The optional qualifier `SHORT` specifies that the section is a
short section. Objects defined there can be accessed using the
direct addressing mode.

Example: The following example demonstrates the definition and usage
             of a `SHORT` section. On line number 12, the symbol `data` is
             accessed using the direct addressing mode.

```
2    2                              dataSec: SECTION  SHORT
3    3    000000                    data: DS.B 1
4    4
5    5       0000 0AFE              initSP:  EQU $AFE
6    6
7    7                              codeSec: SECTION
8    8
9    9                              entry:
10   10   000000 CF 0AFE                    LDS  #initSP
11   11   000003 C600                       LDAB #0
12   12   000005 5Bxx                       STAB data
```

8.33 SET — Set Symbol Value

Syntax: <label>: SET <expression>

Description: Similar to the EQU directive, the SET directive assigns the value of the <expression> in the operand field to the symbol in the <label> field. The <expression> cannot include a symbol that is undefined or not yet defined. The <label> is an assembly time constant; SET does not generate machine code.

The value is temporary; a subsequent SET directive can redefine it.

Example:
```
  2    2              0000 0002    count: SET 2
  3    3    000000 02              loop:  DC.B count
  4    4              0000 0002           IFNE count
  5    5              0000 0001    count: SET count - 1
  6    6                                  ENDIF
  7    7    000001 01                      DC.B count
  8    8              0000 0001           IFNE count
  9    9              0000 0000    count: SET count - 1
 10   10                                   ENDIF
 11   11   000002 00                       DC.B count
 12   12               0000 0000            IFNE count
```

The value associated with the label count is decremented after each DC.B instruction.

8.34 SPC — Insert Blank Lines

Syntax: SPC <count>

Description: Inserts 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.

8.35 TABS — Set Tab Length

Syntax: TABS <n>

Description: Sets tab length to <n> spaces. The default tab length is eight.
<n> may range from 0 to 128.

8.36 TITLE — Provide Listing Title

Syntax: TITLE "title"

Description: Prints the <title> on the head of each 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.

For compatibility with MASM, a title can also be specified
without quotes.

8.37 XDEF — External Symbol Definition

Syntax: XDEF [.<size>] <label>[,<label>]...

where

<size> = B, W (default), or L

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 listed in an XDEF directive is only limited by the memory available at assembly time.

Example:
```
 XDEF Global   ;Global can be referenced in other module
 XDEF AnyCase  ;Note that the linker and assembler are
               ; case sensitive to names.

 GLOBAL: DS.B 4
 ...
     AnyCase NOP
```

8.38 XREF — External Symbol Reference

Syntax: XREF [.<size>] <symbol>[,<symbol>]...

                 where

                 <size> = B, W (default), or L

Description: This directive specifies symbols referenced in the current
                 module but defined in another module. The list of symbols and
                 corresponding 16-bit values are passed to the linker.

                 The number of symbols listed in an XREF directive is only
                 limited by the memory available at assembly time.

Example:
```
XREF OtherGlobal  ; Reference "OtherGlobal" defined in
                  ; another module
```

See also: **8.37 XDEF — External Symbol Definition**

Section 9. Macros

9.1 Contents

9.2 Introduction

This chapter describes the functionality and use of macros for the MCUez HC12 assembler used with an MCUez application.

9.3 Macro Overview

A macro is a template for a code sequence. A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the reference by which the macro is subsequently called. Once a macro is defined, subsequent references to the macro name are replaced by its code sequence.

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 inline coding variations of the macro definition.

Macro calls produce inline 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.

9.4 Defining a Macro

The definition of a macro consists of four parts:

1. Header statement, a `MACRO` directive with a label that names the macro

2. Body of the macro, a sequential list of assembler statements, some possibly including argument placeholders

3. The `ENDM` directive that terminates the macro definition

4. The `MEXIT` directive that stops macro expansion

The body of a macro is a sequence of assembler source statements. Macro parameters are defined by parameter designators within these source statements. Valid macro definition statements include the set of assembly language instructions, assembler directives, and calls to previously defined macros. However, macro definitions may not be nested.

NOTE: *Refer to* **Section 8. Assembler Directives** *for information about the* `MACRO`, `ENDM`, `MEXIT`, *and* `MLIST` *directives.*

9.5 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, all macros must be defined before their 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, separated by commas.

The macro call produces inline 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 macro calls are also expanded at this time.

9.6 Macro Parameters

A maximum of 36 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 back slash 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 (.).

Example:

Consider this macro definition:

```
MyMacro: MACRO
         DC.\0    \1, \2
         ENDM
```

When this macro is used in a program, for instance:

```
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 designators \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 with 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 is assembled inline.

It is possible to specify a null argument in a macro call by inserting a comma with no character (no space character) between the comma and the preceding macro name or comma that follows an argument. When a null argument 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.

9.7 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: *\@ may be preceded or followed by additional characters for clarity and to prevent ambiguity.*

Example:

```
clear: MACRO
         LDX     \1
         LDAB    #16
\@LOOP:  CLR     0,X
         INX
         DECB
         BNE     \@LOOP
       ENDM
```

This macro is called in the application:

```
        clear       temporary
      clear       data
```

The two macro calls of clear are expanded this way:

```
        clear   temporary
              LDX       temporary
              LDAB      #16
  _00001LOOP:CLR       0,X
              INX
              DECB
              BNE       _00001LOOP
          clear   data
              LDX       data
              LDAB      #16
  _00002LOOP:CLR       0,X
              INX
              DECB
              BNE       _00002LOOP
```

9.8 Macro Expansion

When the assembler reads a statement in a source program that calls a previously defined macro, it processes the call as described here.

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 arguments that have not been defined at invocation time are initialized 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.

9.9 Nested Macros

Macro expansion is performed at invocation time, which is also the case for nested macros. If the macro definition contains a nested macro call, the nested macro expansion takes place inline. Recursive macro calls are also supported.

A macro call is limited to the length of one line, for example, 1024 characters.

Section 10. Assembler Listing File

10.1 Content

10.2 Introduction

The assembler listing file is the output file of the assembler, which contains information about the generated code. The listing file is generated when the –L option is activated. If an error is detected during assembly, no listing file is generated.

The amount of information available depends on these assembly options:

   -Li, -Lc, -Ld, -Le

Information in the listing file also depends on the following assembly directives:

   LIST, NOLIST, CLIST, MLIST

The format of the listing file is influenced by these directives:

   PLEN, LLEN, TABS, SPC, PAGE, NOPAGE, TITLE

The name of the generated listing file is <basename>.*lst*.

10.3 Page Header

The page header consists of three lines:

- The first line contains an optional user string defined in the directive TITLE.

- The second line contains the name of the assembler vendor (MOTOROLA) as well as the target processor name (HCxx).

- The third line contains a copyright notice.

Example:

```
Demo Application
Motorola HC12-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997
```

10.4 Source Listing

The following sections describe each column. The source listing is divided into five columns:

1. Abs.

2. Rel.

3. Loc.

4. Obj. code

5. Source line

10.4.1 Absolute (Abs.) Listing

This column contains the absolute line number for each instruction. The absolute line number is the line number in the DBG file, which contains all included files and where all macro calls have been expanded.

Example:

```
Abs. Rel.  Loc.     Obj. code  Source line
---- ----  ------   ---------  -----------
1    1                         ;----------------
2    2                         ; File: test.o
3    3                         ;----------------
4    4
5    5                            INCLUDE "macro.inc"
6    1i                        cpChar: MACRO
7    2i                               LDD \1
8    3i                               STD \2
9    4i                           ENDM
10    5i
11    6                           codeSec: SECTION
12    7                        Start:
13    8                            cpChar ch1, ch2
14    2m   000000 FC xxxx   +      LDD ch1
15    3m   000003 7C xxxx   +      STD ch2
16   9     000006 A7              NOP
17   10    000007 A7              NOP
```

In the previous example, the line number displayed in the column Abs. is incremented for each line.

10.4.2 Relative (Rel.) Listing

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.

An `i` suffix is appended to the relative line number, if 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.

Example:

```
Abs. Rel.  Loc.    Obj. code  Source line
---- ----  ------  ---------  -----------
1    1                        ;-----------------------
2    2                        ; File: test.o
3    3                        ;-----------------------
4    4
5    5                          INCLUDE "macro.inc"
6    1i                       cpChar: MACRO
7    2i                              LDD \1
8    3i                              STD \2
9    4i                            ENDM
10   5i
11   6                        codeSec: SECTION
12   7                        Start:
13   8                            cpChar ch1, ch2
14   2m     000000 FC xxxx        +      LDD ch1
15   3m     000003 7C xxxx        +      STD ch2
16   9      000006 A7            NOP
17   10     000007 A7            NOP
```

In the previous example, the line number displayed in the column Rel. represents the line number of the corresponding instruction in the source file. The `1i` on absolute line number 6 denotes that the instruction `cpChar: MACRO` is located in an included file. The `2m` on absolute line number 14 denotes that the instruction `LDD ch1` is generated by a macro expansion.

10.4.3 Location (Loc.) Listing

This column contains the address of the instruction. For absolute sections, the address is preceded by the letter 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 address is a hexadecimal number coded on six 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, …).

Example:

```
Abs. Rel.  Loc.   Obj. code  Source line
---- ----  ------ ---------  -----------
1    1                           ;-------------------
2    2                           ; File: test.o
3    3                           ;-------------------
4    4
5    5                           INCLUDE "macro.inc"
6    1i                      cpChar: MACRO
7    2i                              LDD \1
8    3i                              STD \2
9    4i                          ENDM
10   5i
11   6                        codeSec: SECTION
12   7                        Start:
13   8                            cpChar ch1, ch2
14   2m    000000 FC xxxx    +       LDD ch1
15   3m    000003 7C xxxx    +       STD ch2
16   9     000006 A7                 NOP
17   10    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 LDD ch1 is located at offset 0 from the codeSec section start address. The instruction STD ch2 is located at offset 3 from the codeSec section start address.

10.4.4 Object (Obj.) Code Listing

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. The address is determined at link time.

Example:

```
Abs. Rel.   Loc.   Obj. code  Source line
---- ----   ------ ---------  -----------
1    1                        ;--------------------
2    2                        ; File: test.o
3    3                        ;--------------------
4    4
5     5                          INCLUDE "macro.inc"
6    1i                      cpChar: MACRO
7    2i                            LDD \1
8    3i                            STD \2
9    4i                          ENDM
10   5i
11   6                      codeSec: SECTION
12   7                      Start:
13   8                            cpChar ch1, ch2
14   2m    000000 FC xxxx    +      LDD ch1
15   3m    000003 7C xxxx    +      STD ch2
16   9     000006 A7                NOP
17   10    000007 A7                NOP
```

10.4.5 Source Line Listing

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 performed.

Example:

```
Abs. Rel.  Loc.    Obj. code   Source line
----  ----  ------  ---------   -----------
1     1                         ;--------------------
2     2                         ; File: test.o
3     3                         ;--------------------
4     4
5     5                            INCLUDE "macro.inc"
6     1i                        cpChar: MACRO
7     2i                             LDD \1
8     3i                             STD \2
9     4i                          ENDM
10    5i
11    6                         codeSec: SECTION
12    7                         Start:
13    8                             cpChar ch1, ch2
14    2m    000000 FC xxxx      +      LDD ch1
15    3m    000003 7C xxxx      +      STD ch2
16    9     000006 A7              NOP
17    10    000007 A7              NOP
```

Section 11. Operating Procedures

11.1 Contents

11.2 Introduction

This section provides operating procedures for the MCUez assembler.

11.2.1 Working with Absolute Sections

An absolute section has its start address known at assembly time. (See modules *fiboorg.asm* and *fiboorg.prm* in the demo directory.)

11.2.2 Defining Absolute Sections in the Assembly Source File

An absolute section is defined by the directive ORG. The macro assembler generates a pseudo section named ORG_<index>, where <index> is an integer which is incremented each time an absolute section is encountered.

Example:

Defining an absolute section containing data:

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

In the previous example code, the label cst1 will be located at address $A00, and label cst2 will be located at address $A01, as shown in this code listing.

```
 1     1                                ORG $A00
 2     2   a000A00 A6         cst1: DC.B    $A6
 3     3   a000A01 BC         cst2: DC.B    $BC
 4     4                                ORG    $800
 5     5   a000800            var:  DS.B    1
```

Defining an absolute section containing code:

```
      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 code, the instruction LDAA will be located at address $C00 and instruction ADDA at address $C03, as shown in this code listing.

```
6    6                        ORG $C00
7    7                 entry:
8    8  a000C00 B6 0A00   LDAA cst1  ; Load value in cst1
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
```

To avoid problems during linking or executing an application, an assembly file should at least:

- Initialize the stack pointer (using the instruction LDS).

- Publish the application entry point using XDEF.

- The programmer should ensure that the addresses specified in the source file are valid addresses for the MCU being used.

11.2.3 Linking an Application Containing Absolute Sections

Applications containing only absolute sections must be linked.

A linker parameter file must contain at least:

- Name of the absolute file

- Name of the object file which should be linked

- Specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.

- 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.

- Specification of the application entry point

- Definition of the reset vector

The minimal linker parameter file will look like this:

```
LINK test.abs    /* Name of the executable file generated. */
NAMES
  test.o         /* Name of the object files in the application. */
END
SEGMENTS
/* 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  0x1000 TO 0x1FFF;
/* READ_WRITE memory area. There should be no overlap between this memory
area and the absolute sections defined in the assembly source file. */
  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  DEFAULT_RAM    INTO MY_RAM;
/*Relocatable code and constant sections are allocated in MY_ROM.*/
  DEFAULT_ROM    INTO MY_ROM;
END
INIT entry       /*Application entry point.*/
VECTOR ADDRESS 0xFFFE entry /*Initialization of the reset vector.*/
```

> ***NOTE:*** *Allow no overlap between the absolute section defined in the assembly source file and the memory area defined in the PRM file.*

> ***NOTE:*** *Since the memory areas (segments) specified in the PRM file are used only to allocate relocatable sections, nothing will be allocated there if the application contains only absolute sections.*

The module *fiboorg.asm* located in the demo directory is a small example of using absolute sections in an application.

11.3 Working with Relocatable Sections

A relocatable section is a section whereby the start address is determined at link time. See modules *fibo.asm* and *fibo.prm* in the demo directory.

11.3.1 Defining Relocatable Sections in the Assembly Source File

A relocatable section is defined using the directive SECTION.

Example:

Defining a relocatable section 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 offset 0 from the section constSec start address, and label cst2 will be located at offset 1 from the section constSec start address.

```
1    1                          constSec: SECTION
2    2    000000 A6             cst1: DC.B    $A6
3    3    000001 BC             cst2: DC.B    $BC
4    4
5    5                          dataSec:  SECTION
6    6    000000               var: DS.B    1
```

Defining a relocatable section containing code:

```
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 will be located at offset 3 from the codeSec start address.

```
8    8                           codeSec:  SECTION
9    9                           entry:
10   10   000000 B6 xxxx LDAA cst1    ; Load value in cst1
11   11   000003 BB xxxx ADDA cst2    ; Add value in cst2
12   12   000006 7A xxxx STAA var     ; Store in var
13   13   000009 20F5    BRA entry
```

To avoid problems during linking or executing an application, an assembly file must:

- Initialize the stack pointer using the instruction LDS

- Publish the application entry point using XDEF

11.3.2 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

- Name of the object file which should be linked

- Specification of a memory area where the sections containing variables must be allocated

- Specification of a memory area where sections containing code or constants must be allocated

- Specification for the application entry point

- Definition of the reset vector

The minimal linker parameter file will look like this:

```
LINK test.abs /* Name of the executable file generated. */
NAMES
  test.o      /* Name of the object files in the application. */
END
SEGMENTS
/* 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. */
```

> **NOTE:** *The programmer should ensure that the memory ranges specified in the SEGMENT block are valid addresses for the MCU being used.*

The module *fibo.asm* located in the demo directory is a small example of using the relocatable sections in an application.

11.4 Initializing the Vector Table

The vector table is initialized in the assembly source file or in the linker parameter file. Initializing it in the PRM file is recommended.

11.4.1 Initializing the Vector Table in the Linker PRM File

Initializing the vector table from the PRM (parameter) file allows initialization of single entries in the table. The user can decide to initialize all the entries in the vector table or not.

The labels or functions inserted in the vector table must also be implemented in the assembly source file. All these labels must be published; otherwise, they cannot be addressed in the linker PRM file.

Example:

```
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 appropriate element in the table
INC  0, X            ; The table element is incremented
            RTI
entry:
            LDS #$AFE
loop:       BRA loop
```

NOTE: *The functions* `XIRQFunc`, `SWIFunc`, *and* `ResetFunc` *are published. This is required because they are referenced in the linker PRM file. All interrupt functions must be terminated with an RTI instruction.*

The vector table is initialized using the linker command VECTOR ADDRESS.

Example:

```
LINK test.abs
NAMES
  test.o
END

SEGMENTS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0B00 TO 0x0CFF;
END
PLACEMENT
  .data        INTO MY_RAM;
  .text        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. The statement* `VECTOR ADDRESS 0xFFF4 XIRQFunc` *specifies that the address of function* `XIRQFunc` *should be written at address* `0xFFF4`.

11.4.2 Initializing Vector Table in Assembly Source Files Using a Relocatable Section

Initializing the vector table in the assembly source file requires that all entries in the table are initialized. Interrupts that 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 assembler 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.

Example:

```
                          XDEF ResetFunc
DataSec: SECTION
Data:DS.W 5                ;Each interrupt increments an element of 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 section `VectorTable` is defined as a word (2-byte constant) because the entries in the HC12 vector table are 16 bits wide. In the previous example, the constant `XIRQInt` is initialized with the address of the label `XIRQFunc`. The constant `COPResetInt` is initialized with the address of the label `DummyFunc` because this interrupt is not in use. All labels specified as initialization values must be defined, published (using XDEF), or imported (using XREF) before the vector table section.*

The section should now be placed at the expected address. This is performed in the linker parameter file.

Example:

```
LINK test.abs
NAMES  test.o END

SEGMENTS
  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.*
> *The statement* `VectorTable INTO Vector` *specifies that the vector table 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`*. 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.*

11.4.3 Initializing the Vector Table in the Assembly Source File Using an Absolute Section

Initializing the vector table in the assembly source file requires that all entries in the table are initialized. Interrupts that are not used must be associated with a standard handler.

The labels or functions that are 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.

Example:

```
        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
```

NOTE: *Each constant in the section* `VectorTable` *is defined as a word (2-byte constant, because the entry in the HC12 vector table is 16 bits wide. In the previous example, the constant* `IRQInt` *is initialized with the address of the label* `IRQFunc`, *the constant* `COPResetInt` *is initialized with the address of the label* `DummyFunc`, *etc. Labels specified as initialization values must be defined, published (using* `XDEF`) *or imported (using* `XREF`) *before the vector table section. The statement* `ORG $FFF2` *specifies that the following section must start at address* `$FFF2`.

The section should now be placed at the expected address. This is performed in the linker parameter file.

Example:

```
LINK test.abs
NAMES
  test.o
END

SEGMENTS
  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 referenced objects in the absolute file.*

11.5 Splitting an Application into Different Modules

A complex application or application involving several programmers can be split into several simple modules. In order to avoid any problem when merging different modules, adhere to the following.

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 assembly file (*Test1.asm*):

```
        XDEF AddSource
        XDEF Source

initStack:EQU $AFF

DataSec: SECTION
Source:   DS.W 1
CodeSec:  SECTION
AddSource:
        ADD Source
        STD Source
        RTS
```

Corresponding include file (*Test1.inc*):

```
 XREF AddSource
 XREF AddSource
; The function AddSource adds the value stored in the variable
; Source to the content of register D. The result of the computation
; is stored in the variable Source.
;
; Input Parameter : register D contains the value, which should be
;                   added to the variable Source.
; Output Parameter: register D contains the result of the addition.

        XREF Source
; The variable Source is a word variable.
```

Each assembly module using a symbol defined in another assembly file should include the corresponding include file.

Example of assembly file (*Test2.asm*):

```
            XDEF entry
            INCLUDE "Test1.inc"

    initStack: EQU $AFE

    CodeSec:    SECTION
    entry:      LDS #initStack
                LDD #$7
                JSR AddSource
                BRA entry
```

The application PRM file must list both object files used to build the application. When a section is present in the different object files, the object file sections are concatenated in 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 PRM file (*Test2.prm*):

```
LINK test2.abs  /* Name of executable file generated. */

NAMES
  test1.o test2.o /*Name of object files building the application.*/
END

SEGMENTS
  MY_ROM  = READ_ONLY  0x0B00 TO 0x0BFF; /* READ_ONLY memory area */
  MY_RAM  = READ_WRITE 0x0800 TO 0x08FF; /* READ_WRITE memory area */
END

PLACEMENT
  DataSec, .data INTO MY_RAM; /*variables are allocated in MY_RAM */
  CodeSec, .text INTO MY_ROM; /* code and constants are allocated in
                         MY_ROM*/
END

INIT entry        /* Definition of the application entry point. */

VECTOR ADDRESS 0xFFFE entry/* Definition of the reset vector. */
```

NOTE: *The statement* `NAMES test1.o test2.o END` *lists the two object files building the application. A space character separates the object filenames. The section* `CodeSec` *is defined in both object files. In* `test1.o`, *the section* `CodeSec` *contains the symbol* `AddSource`. *In* `test2.o`, *the section* `CodeSec` *contains the symbol* `entry`. *According to the order in which the object files are listed in the* `NAMES` *block, the function* `AddSource` *will be allocated first (at address 0xB00) and symbol* `entry` *will be allocated next to it.*

11.6 Using Direct Addressing Mode to Access Symbols

The different methods to inform the assembler it should use the direct addressing mode on a symbol are discussed here.

11.6.1 Using Direct Addressing Mode to Access External Symbols

External symbols that should be accessed using the direct addressing mode, must be declared by the directive `XREF.B`.

Example:

```
    XREF.B ExternalDirLabel
    XREF   ExternalExtLabel
…
    LDD    ExternalDirLabel      ; Direct addressing mode is used
…
    LDD    ExternalExtLabel      ; Extended addressing mode is used
```

11.6.2 Using Direct Addressing Mode to Access Exported Symbols

Symbols that are exported using the directive `XDEF.B`, will be accessed by the direct addressing mode. Symbols that are exported using the directive `XDEF`, are accessed using the extended addressing mode.

Example:

```
    XDEF.B DirLabel
    XDEF   ExtLabel
…
    LDD    DirLabel ; Direct addressing mode is used
…
    LDD    ExtLabel ; Extended addressing mode is used
```

11.6.3 Defining Symbols in the Direct Page

Symbols that are defined in the predefined section BSCT are always accessed using the direct addressing mode.

Example:

```
…
      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
```

11.6.4 Using a Force Operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode.

The supported force operators are:

- < or .B to force direct addressing mode

- > or .W to force extended addressing mode

Example:

```
…
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
```

11.6.5 Using SHORT Sections

Symbols defined in a section with the qualifier SHORT are always accessed using the direct addressing mode.

Example:

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

11.7 Directly Generating an .abs File

The MCUez assembler generates an *.abs* file directly from an assembly source file. A Motorola S file is generated at the same time and can be directly burnt into an EPROM.

11.7.1 Assembler Source File

When an *.abs* file is generated using the assembler (no linker), the application must be implemented in a single assembly unit and contain only absolute sections. This is shown in this code example.

Example:

```
        ABSENTRY entry ; Specifies the application
                       ; Entry point
iniStk: EQU $AFE       ; Initial value for SP
        ORG $FFFE      ; Reset vector definition
Reset:  DC.W entry
        ORG $40        ; Define an absolute constant section
var1:   DC.B 5         ; Assign 5 to the symbol var1
        ORG $80        ; Define an absolute data section
data:   DS.B 1         ; Define one byte variable in RAM
                       ; address 40
        ORG $B00       ; Define an absolute code section
entry:
        LDS  #iniStk; Load stack pointer
        LDAA  var1
main:
        INCA
        STAA  data
        BRA   main
```

When writing an assembly source file for direct absolute file generation, pay special attention to these points:

- The directive ABSENTRY is used to write the entry point address in the generated absolute file. To set the entry point of the application to the label entry in the absolute file, this code is needed:

```
        ABSENTRY entry
```

- The reset vector must be initialized in the assembly source file, specifying the application entry point. An absolute section is created at the reset vector address. This section contains the application entry point address.

  To set the entry point of the application at address $FFFE to the label entry, this code is needed:

```
        ORG $FFFE     ; Reset vector definition
 Reset:DC.W entry
```

- It is strongly recommended to use separate sections for code, data, and constants. All sections used in the assembler application must be absolute. They must be defined using the ORG directive. The address for constant or code sections has to be located in the ROM memory area, while the data sections have to be located in the RAM area (according to the hardware used). The programmer must ensure that no sections overlap.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

11.7.2 Assembling and Generating the Application

Once the source file is available, it can be assembled.

1. Start the macro assembler by clicking the **eZASM** icon in the **MCUez Shell** toolbar. The assembler is started as shown in **Figure 11-1**. Enter the name of the file to be assembled in the editable combo box, for example, *abstest.asm*.
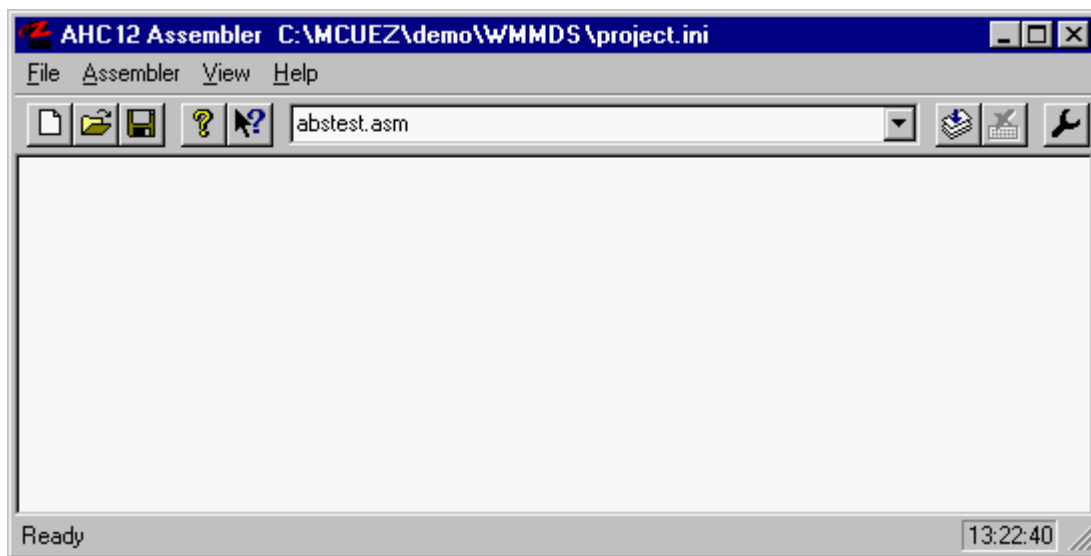
Figure 11-1. Starting the MCUez Assembler

2. Select the menu entry **Assembler | Options**. The **Options Settings** dialog is displayed, as shown in **Figure 11-2**.

.
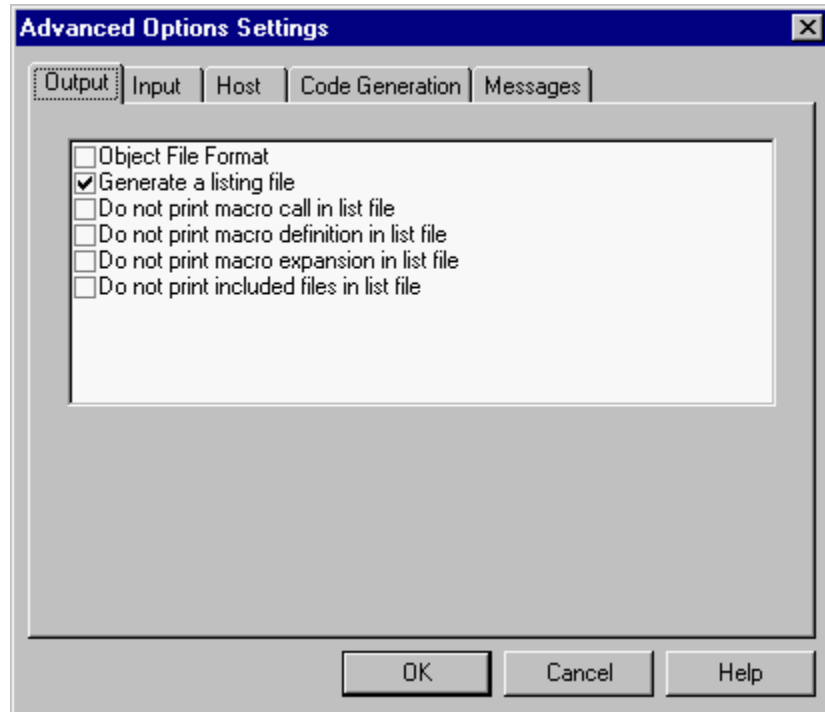
Figure 11-2. Options Setting Dialog Box

3. In the **Output** folder, select the check box in front of the label **Object File Format**. More information is displayed at the bottom of the dialog, as shown in **Figure 11-3**.
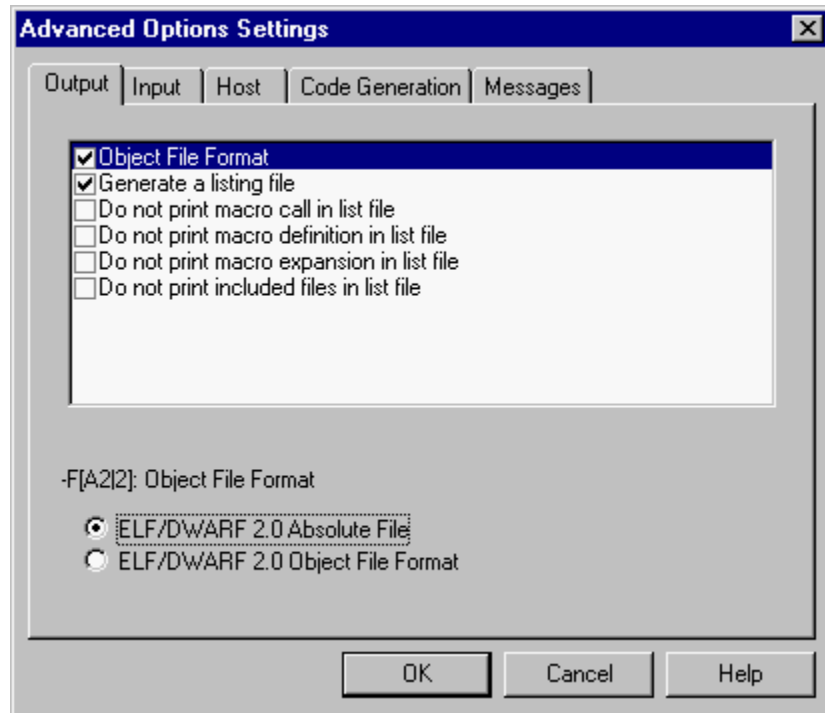
Figure 11-3. Selecting the Object File Format

4. Select the radio button **ELF/DWARF 2.0 Absolute File** and click **OK**. The assembler is now ready to generate an absolute file. Click on the **Assemble** button to assemble the file. The assembly process is shown in **Figure 11-4**.
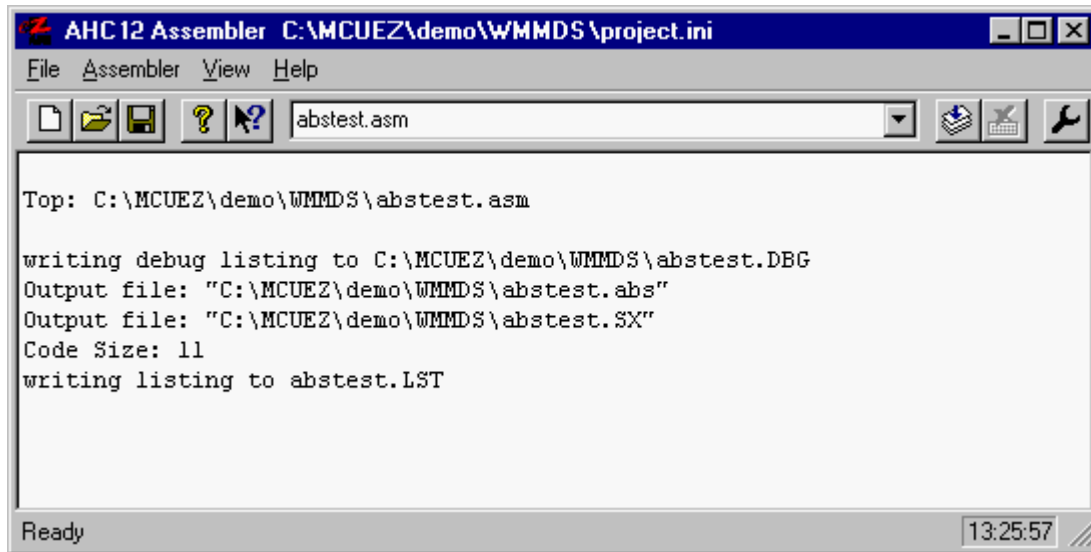
Figure 11-4. Generating an .abs File

The generated absolute file (*.abs*) is used with the target board or emulator. This file can be downloaded directly to the HC08 target. The target must be reset from the menu option **MMDS0508 | Reset** before running the application. The *.sx* file that is generated is a standard Motorola S record file. This file can be directly burnt into an EPROM.

User's Manual — MCUez HC12 Assembler

Section 12. Assembler Messages

12.1 Contents

For More Information On This Product,
Go to: www.freescale.com

12.2 Introduction

The assembler can generate three types of messages:

1. Warning

2. Error

3. Fatal

12.2.1 Warning

A message will be printed and assembly will continue. Warning messages are used to indicate possible programming errors to the user.

12.2.2 Error

A message will be printed and assembly will stop. Error messages are used to indicate illegal usage of the language.

12.2.3 Fatal

A message will be printed and assembly will be aborted. A fatal message indicates a severe error that will stop the assembly process.

12.3 Message Codes

If the assembler prints out a message, the message contains a message code (A for assembler) and a 4- to 5-digit number. This number may be used to search for the indicated message in the manual. All messages generated by the assembler are documented in increasing order for easy and fast retrieval.

Each message also has a description and, if available, a short example with a possible solution or tips to fix a problem. For each message, the type of message is also noted. For instance, error indicates an error message.

12.3.1 A1000: Conditional Directive not Closed

Type: Error

Description: One of the conditional blocks is not closed. A conditional block can be opened using one of these directives:

IF, IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE, IFC, IFNC, IFDEF, IFNDEF

Example:

```
        IFEQ (defineConst)
const1:  DC.B 1
const2:  DC.B 2
```

Tip: Close the conditional block with an ENDIF or ENDC directive.

Example:

```
        IFEQ (defineConst)
const1:  DC.B 1
const2:  DC.B 2
        ENDIF
```

Be careful: A conditional block, which starts inside a macro, must be closed within the same macro.

Example: The following portion of code generates an error, because the conditional block IFEQ is opened within the macro MyMacro and is closed outside the macro.

```
MyMacro: MACRO
        IFEQ (SaveRegs)
            NOP
            NOP
            ENDM
            NOP
        ENDIF
```

12.3.2 A1001: Conditional Else not Allowed Here

Type: Error

Description: A second ELSE directive is detected in a conditional block.

Example:

```
        IFEQ (defineConst)
...
  ELSE
...
  ELSE
...
  ENDIF
```

Tip: Remove the superfluous ELSE directive.

Example:

```
        IFEQ (defineConst)
...
  ELSE
...
  ENDIF
```

12.3.3 A1051: Zero Division in Expression

Type: Error

Description: A zero division is detected in an expression.

Example:

```
label:  EQU 0;
label2: EQU $5000
...
        LDX #(label2/label)
```

Tip: Modify the expression or specify it in a conditional assembly block.

Example:

```
label:  EQU 0;
label2: EQU $5000
...
    IFNE (label)
        LDX #(label2/label)
    ELSE
        LDX #label2
    ENDIF
```

12.3.4 A1052: Right Parenthesis Expected

Type: Error

Description: A right parenthesis is missing in an assembly expression or in an expression containing the PAGE operator.

Example:

```
MyData: SECTION
variable: DS.B 1

label:  EQU (2*4+6
label2: EQU PAGE (variable
```

Tip: Insert the right parenthesis at the correct position.

Example:

```
MyData: SECTION
variable: DS.B 1

label:  EQU (2*4)+6
label2: EQU PAGE(variable)
```

12.3.5 A1053: Left Parenthesis Expected

Type: Error

Description: A left parenthesis is missing in an expression containing a
                 reference to the page (bank) where an object is allocated.

Example:

```
MyData: SECTION
variable: DS.B 1
label3: EQU PAGE variable)
```

Tip: Insert the left parenthesis at the correct position.

Example:

```
MyData: SECTION
variable: DS.B 1

label3: EQU PAGE (variable)
```

12.3.6 A1054: References on Non-Absolute Objects Are not Allowed When Options -FA1 or -FA2 Are Enabled

Type: Error

Description: A reference to a relocatable object has been detected during generation of an absolute file by the assembler.

Example:

```
                XREF extData
DataSec: SECTION
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
         LDX extData
```

Tips: When generating an absolute file, the application should be encoded in a single source file and should contain only a relocatable symbol.

To avoid this message, define all sections as absolute sections and remove all XREF directives from the source file.

Example:

```
         ORG $B00
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
```

12.3.7 A1101: Illegal Label: Label is Reserved

Type: Error

Description: A reserved identifier is used as a label. Reserved identifiers are:

- Mnemonics associated with target processor registers are A, B, CCR, D, PC, SP, TEMP2, TEMP3, X, Y.

- Mnemonics associated with special target processor operator are PAGE.

Example:

```
A:    NOP
      NOP
      RTS
```

Tip: Modify the label name to an identifier that is not reserved.

Example:

```
ASub: NOP
      NOP
      RTS
```

MCUez HC12 Assembler

12.3.8 A1103: Illegal Redefinition of Label

Type: Error

Description: The label specified in front of a comment, assembly
               instruction, or directive is detected twice in a source file.

Example:

```
DataSec1: SECTION
label1: DS.W 2
label2: DS.L 2
…
CodeSec1: SECTION
Entry:  LDS #$4000
        LDX #label1
        CPX #$500
        BNE label2
…
label2: RTS
```

Tip: Modify the label names so that labels are unique in each
               assembly file.

Example:

```
DataSec1: SECTION
DataLab1: DS.W 2
DataLab2: DS.L 2
…
CodeSec1: SECTION
Entry:  LDS #$4000
        LDX #label1
        CPX #$500
        BNE label2
…
CodeLab2: RTS
```

12.3.9 A1104: Undeclared User-Defined Symbol <symbolName>

Type: Error

Description: The label <symbolName> is referenced in the assembly file, but it is never defined.

Example:

```
Entry:
  LDX #56
  STX #Variable
  RTS
```

Tip: The label <symbolName> must be defined in the current assembly file or specified as an external label.

Example:

```
XREF  Variable
…
Entry:
  LDX #56
  STX #Variable
  RTS
```

12.3.10 A1201: Label <labelName> Referenced in Directive ABSENTRY is not Defined in Code Segment

Type: Error

Description: The label specified in the directive ABSENTRY is an EQU label or is located in a data section. The label specified in ABSENTRY must be a valid label defined in a code section. Only labels defined in a code segment are allowed in the ABSENTRY directive.

Example:

```
          ABSENTRY  const
const     EQU $67
```

Tip: Specify a label defined in a code section or remove the directive ABSENTRY.

Example:

```
          ABSENTRY  entry
          ORG $300
entry     NOP
          NOP
          NOP
```

12.3.11 A2301: Label is Missing

Type: Error

Description: A label is missing at the front of an assembly directive requiring a label: SECTION, EQU, and SET.

Example:

```
SECTION 4
…
EQU $67
…
SET $77
```

Tip: Insert a label in front of the directive.

Example:

```
codeSec: SECTION 4
   …
myConst: EQU $67
   …
mySetV:  SET $77
```

12.3.12 A2302: Macro Name is Missing

Type: Error

Description: A label name is missing in front of a MACRO directive.

Example:

```
MACRO
  LDD \1
  ADD \2
  STD \1
ENDM
```

Tip: Insert a label in front of the MACRO directive.

Example:

```
AddM: MACRO
     LDD \1
     ADD \2
     STD \1
    ENDM
```

12.3.13 A2303: ENDM is Illegal

Type: Error

Description: An ENDM directive is detected outside a macro.

Example:

```
AddM: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
    NOP
    AddM data1, data2
    ENDM
```

Tip: Remove the superfluous ENDM directive.

Example:

```
AddM: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
    NOP
    AddM data1, data2
```

12.3.14 A2304: Macro Definition Within Definition

Type: Error

Description: A macro definition is detected inside another macro definition.
                    The macro assembler does not support this.

Example:

```
AddM: MACRO
AddX: MACRO
        LDX \1
        INX
        STX \1
    ENDM
        LDD \1
        ADD \2
        STD \1
    ENDM
```

Tip: Define the second macro outside the first one.

Example:

```
AddX: MACRO
        LDX \1
        INX
        STX \1
    ENDM
AddM: MACRO
        LDD \1
        ADD \2
        STD \1
    ENDM
```

12.3.15 A2305: Illegal Redefinition of Instruction or Directive Name

Type: Error

Description: An assembly directive or an HC12 instruction name has been
used as a macro name. This is not allowed to avoid any
ambiguity when the symbol name is encountered. The macro
assembler cannot detect if the symbol refers to the macro or the
instruction.

Example:

```
ADDD: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
```

Tip: Change the name of the macro to an unused identifier.

Example:

```
ADDM: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
```

12.3.16 A2306: Macro not Closed at End of Source

Type: Error

Description: An ENDM directive is missing at the end of a macro. The end
                   of the input file is detected before the end of the macro.

Example:

```
AddM: MACRO
        LDD  \1
        ADD  \2
        STD  \1
      NOP
      AddM data1, data2
```

Tip: Insert the missing ENDM directive at the end of the macro.

Example:

```
AddM: MACRO
        LDD  \1
        ADD  \2
        STD  \1
        ENDM
      NOP
      AddM data1, data2
```

12.3.17 A2307: Macro Redefinition

Type: Error

Description: The input file contains the definition of two macros that have
                the same name.

Example:

```
AddM: MACRO
        LDX \1
        INX
        STX \1
      ENDM
   ...
AddM: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
```

Tip: Change the name of one of the macros to generate unique
                identifiers.

Example:

```
AddX: MACRO
        LDX \1
        INX
        STX \1
      ENDM
AddM: MACRO
        LDD \1
        ADD \2
        STD \1
      ENDM
```

12.3.18 A2308: Filename Expected

Type: Error

Description: A filename is expected in an INCLUDE directive.

Example:

```
xxx: EQU  $56
     …
     INCLUDE xxx
```

Tip: Specify a filename after the include directive.

Example:

```
xxx: EQU  $56
     …
     INCLUDE "xxx.inc"
```

12.3.19 A2309: File not Found

Type: Error

Description: The macro assembler cannot locate a file that is specified in the INCLUDE directive.

Tip: If the file exists, check if the directory is specified in the GENPATH environment variable. First check if the project directory is correct and if the *default.env* file is present. The macro assembler looks for the include files in the project directory, then in the directory listed in the GENPATH environment variable. If the file does not exist, create it or remove the include directive.

12.3.20 A2310: Illegal Size Character

Type: Error

Description: An invalid size specification character is detected in a DCB, DC, DS, FCC, FCB, FDB, RMB, XDEF, or XREF directive.

For XDEF and XREF directives, valid size specification characters are:

- `.B` for symbols located in a section where direct addressing mode can be used

- `.W` for symbols located in a section where extended addressing mode must be used

For DCB, DC, DS, FCC, FCB, FDB, and RMB directives, valid size specification characters are:

- `.B` for byte variables

- `.W` for word variables

- `.L` for long variables

Example:

```
DataSec: SECTION
label1: DS.Q 2
…
ConstSec: SECTION
label2: DC.I 3, 4, 6
```

Tip: Change the size specification character to a valid one.

Example:

```
DataSec: SECTION
label1: DS.L 2
…
ConstSec: SECTION
label2: DC.W 3, 4, 6
```

12.3.21 A2311: Symbol Name Expected

Type: Error

Description: A symbol name is missing after an XDEF, XREF, IFDEF, or IFNDEF directive.

Example:

```
XDEF $5645
          XREF ; This is a comment
  CodeSec: SECTION
  …
          IFDEF $5634
```

Tip: Insert a symbol name at the requested position.

Example:

```
XDEF exportedSymbol
     XREF importedSymbol; This is a comment
CodeSec: SECTION
…
        IFDEF changeBank
```

12.3.22 A2312: String Expected

Type: Error

Description: A character string is expected at the end of an FCC, IFC, or IFNC directive.

Example:

```
expr:    EQU $5555
expr2:   EQU 5555
DataSec: SECTION
label:   FCC expr
         …
CodeSec: SECTION
         …
         IFC expr, expr2
```

Tip: Insert a character string at the requested position.

Example:

```
expr:    EQU $5555
expr2:   EQU 5555
DataSec: SECTION
label:   FCC "This is a string"
         …
```

12.3.23 A2313: Nesting of Include Files Exceeds 50

Type: Error

Description: The maximum number of nested include files has been exceeded. The assembler supports up to 50 nested include files.

Tip: Reduce the number of nested include files to 50.

12.3.24 A2314: Expression Must Be Absolute

Type: Error

Description: An absolute expression is expected at the specified position.

Assembler directives expecting an absolute value are:

OFFSET, ORG, ALIGN, SET, BASE, DS, LLEN, PLEN, SPC, TABS, IF, IFEQ, IFNE, IFLE, IFLT, IFGE, IFGT.

The first operand in a DCB directive must be absolute.

Example:

```
DataSec: SECTION
label1:  DS.W 1
label2:  DS.W 2
label3:  EQU  8
…
codeSec: SECTION
…
         BASE label1
…
         ALIGN label2
```

Tip: Specify an absolute expression at the specified position.

Example:

```
DataSec: SECTION
label1:  DS.W 1
label2:  DS.W 2
label3:  EQU  8
…
codeSec: SECTION
…
         BASE label3
…
         ALIGN 4
```

12.3.25 A2316: Section Name Required

Type: Error

Description: A SWITCH directive is not followed by a symbol name.
             Absolute expressions or strings are not allowed in a SWITCH
             directive.

             The symbol specified in a SWITCH directive must refer to a
             previously defined section.

Example:
```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
        SWITCH $A344
…
```

Tip: Specify the name of a previously defined section in the
             SWITCH instruction.

Example:
```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
        SWITCH dataSec
…
```

12.3.26 A2317: Illegal Redefinition of Section Name

Type: Error

Description: The name associated with a section was previously used as a label in a code or data section or is specified in an XDEF directive.

The macro assembler does not allow a section name to be exported or to use the same name for a section and a label.

Example:

```
dataSec: SECTION
secLabel: DS.W 3
…
secLabel: SECTION
         LDD secLabel
…
```

Tip: Change section name to a unique identifier.

Example:

```
dataSec: SECTION
seclabel: DS.W 3
…
sec_Label: SECTION
         LDD secLabel
…
```

12.3.27 A2318: Section not Declared

Type:	Error
Description:	The label specified in a SWITCH directive is not associated with a section.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
        SWITCH daatSec
…
```

Tip: Specify the name of a previously defined section in the SWITCH instruction.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
        SWITCH dataSec
…
```

12.3.28 A2320: Value too Small

Type: Error

Description: The absolute expression specified in a directive is too small.

This message can be generated if:

- The expression specified in an ALIGN, DCB, or DS directive is smaller than 1.

- The expression specified in a PLEN directive is smaller than 10. A header is generated on the top of each page from the listing file. This header contains at least six lines. So a page length smaller than 10 lines is not feasible.

- The expression specified in an LLEN, SPC, or TABS directive is smaller than 0 (negative).

Example:

```
                PLEN    5
                LLEN   -4
        dataSec: SECTION
                ALIGN   0
        …
        label1: DS.W    0
        …
```

Tip: Modify the absolute expression to a value in the range specified in the explanation in **12.3.29 A2321: Value too Big**.

Example:

```
                PLEN   50
                LLEN   40
        dataSec: SECTION
                ALIGN  8
        …
        label1: DS.W    1
        …
```

12.3.29 A2321: Value too Big

Type: Error

Description: The absolute expression specified in a directive is too big.

This message can be generated in the following cases:

- The expression specified in an ALIGN directive is bigger than 32,767.

- The expression specified in a DS or DCB directive is bigger than 4096.

- The expression specified in a PLEN directive is bigger than 10,000.

- The expression specified in a LLEN directive is bigger than 132.

- The expression specified in a SPC directive is bigger than 65.

- The expression specified in a TABS directive is bigger than 128.

Example:

```
            PLEN    50000
            LLEN    200
dataSec: SECTION
            ALIGN   40000
…
label1: DS.W    5000
…
```

Tip: Modify the absolute expression to a value in the range specified in the bulleted list here.

Example:

```
            PLEN    50
            LLEN    40
dataSec: SECTION
            ALIGN   8
…
label1: DS.W    1
…
```

12.3.30 A2323: Label is Ignored

Type: Warning

Description: A label is specified in front of a directive that does not accept a label. The macro assembler ignores such labels. These labels cannot be referenced anywhere in the application.

Labels will be ignored in front of these directives:

ELSE, ENDIF, END, ENDM, INCLUDE, CLIST, ALIST, FAIL, LIST, MEXIT, NOLIST, NOL, OFFSET, ORG, NOPAGE, PAGE, LLEN, PLEN, SPC, TABS, TITLE, TTL.

Example:

```
CodeSec: SECTION
         LDD #$5444
label:   PLEN 50
…
label2:  LIST
…
```

Tip: Remove the label that is not required. If a label is needed at that position in a section, define the label on a separate line.

Example:

```
CodeSec: SECTION
         LDD #$5444
label:
         PLEN 50
…
label2:
         LIST
…
```

12.3.31 A2324: Illegal Base (2, 8, 10, 16)

Type: Error

Description: An invalid base number follows a BASE directive. Valid base numbers are 2, 8, 1, or 16. The expression specified in a BASE directive must be an absolute expression and must match one of the values listed here.

Example:

```
                BASE  67
        …
        dataSec: SECTION
        label:   DS.B 8
        …
                BASE  label
```

Tip: Specify a valid value in the BASE directive.

Example:

```
                BASE  16
        …
        dataSec: SECTION
        label:   EQU 8
        …
                BASE  label
```

12.3.32 A2325: Comma or Line End Expected

Type: Error

Description: An incorrect syntax has been detected in a DC, FCB, FDB, XDEF, PUBLIC, GLOBAL, XREF, or EXTERNAL directive. This error message is generated when the values listed in one of the directives are not terminated by an end of line character or when they are not separated by a comma (,) character.

Example:

```
        XDEF  aa1 aa2 aa3 aa4
        XREF  bb1, bb2, bb3, bb4    This is a comment
    …
    dataSec: SECTION
    dataLab1: DC.B 2 | 4 | 6 | 8
    dataLab2: FCB  45, 66, 88     label3:DC.B 4
```

Tip: Use the comma character as a separator between the different items in the list or insert an end of line character at the end of the list.

Example:

```
        XDEF  aa1, aa2, aa3, aa4
        XREF  bb1, bb2, bb3, bb4    ;This is a comment
    …
    dataSec: SECTION
    dataLab1: DC.B 2, 4, 6, 8
    dataLab2: FCB  45, 66, 88
    label3:   DC.B 4
```

12.3.33 A2326: Label is Redefined

Type: Error

Description: A label redefinition has been detected. This message is issued when:

- The label specified in front of a DC, DS, DCB, or FCC directive is already defined.

- One of the label names listed in an XREF directive is already defined.

- The label specified in front of an EQU directive is already defined.

- The label specified in front of a SET directive is already defined and is not associated with another SET directive.

- A label with the same name as an external referenced symbol is defined in the source file.

Example:

```
DataSec: SECTION
label1:  DS.W   4
…
         BSCT
label1: DS.W    1
```

Tip: Modify the source code to use unique identifiers.

Example:

```
DataSec: SECTION
data_label1:  DS.W   4
…
         BSCT
bsct_label1: DS.W    1
```

12.3.34 A2327: ON or OFF Expected

Type: Error

Description: The syntax for an MLIST or CLIST directive is not correct.
These directives expect a unique operand, which take the value
ON or OFF.

Example:

```
CodeSec: SECTION
         …
         CLIST
         …
```

Tip: Specify either ON or OFF after the MLIST or CLIST directive.

Example:

```
CodeSec: SECTION
         …
         CLIST ON
         …
```

12.3.35 A2328: Value is Truncated

Type: Warning

Description: The size of one of the constants listed in a DC directive is
bigger than the size specified in the DC directive.

Example:

```
DataSec: SECTION
cst1:    DC.B  $56, $784, $FF
cst2:    DC.w  $56, $784, $FF5634
```

Tip: Reduce the value of the constant to the size specified in the DC
directive.

Example:

```
DataSec: SECTION
cst1:    DC.B  $56, $7, $84, $FF
cst2:    DC.W  $56, $784, $FF, $5634
```

12.3.36 A2329: FAIL Found

Type: Error

Description: The FAIL directive followed by a number smaller than 500 has been detected in the source file. This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:

```
cpChar: MACRO
        IFC "\1", ""
          FAIL 200   ; Error
          MEXIT
        ELSE
          LDD \1
        ENDIF

        IFC "\2", ""
          FAIL 600   ; Warning
        ELSE
          STD \2
        ENDIF
      ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

12.3.37 A2330: String is not Allowed

Type: Error

Description: A string has been specified as the initial value in a DCB directive. The initial value for a constant block can be any byte, word, or long absolute expression as well as a simple relocatable expression.

Example:

```
CstSec: SECTION
label:  DCB.B  10, "aaaaa"
...
```

Tip: Specify the ASCII code associated with the characters in the string as the initial value.

Example:

```
CstSec: SECTION
label:  DCB.B  5, $61
...
```

12.3.38 A2332: FAIL Found

Type: Warning

Description: The FAIL directive followed by a number bigger than 500 has been detected in the source file. This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:

```
cpChar: MACRO
        IFC "\1", ""
          FAIL 200   ; Error
          MEXIT
        ELSE
          LDD \1
        ENDIF

        IFC "\2", ""
          FAIL 600   ; Warning
        ELSE
          STD \2
        ENDIF
      ENDM
codeSec: SECTION
Start:
        cpChar char1
```

12.3.39 A2333: Forward Reference not Allowed

Type: Error

Description: A forward reference has been detected in an EQU instruction.
                This is not allowed.

Example:

```
CstSec: SECTION
label: DCB.B  10, $61
equLab: EQU label2
...
label2: DC.W $6754
…
```

Tip: Move the EQU after the definition of the label it refers to.

Example:

```
CstSec: SECTION
label:  DCB.B  10, $61
...
label2: DC.W $6754
…
equLab: EQU label2 + 1
```

12.3.40 A2334:Only Labels Defined in the Current Assembly Unit Can Be Referenced in an EQU Expression

Type: Error

Description: One of the symbols specified in an EQU expression is an external symbol, which was previously specified in an XREF directive. This is not allowed due to a limitation in the *ELF* file format.

Example:

```
        XREF label
CstSec: SECTION
lab: DC.B 6
...
equLabel: EQU label+6
...
```

Tip: An EQU label containing a reference to an object must be defined in the same assembly module as the object they refer to. Then the EQU label can be exported to other modules in the application.

Example:

```
        XDEF label, equlabel
        ...
CstSec: SECTION
lab:   DC.B 6
label: DC.W 6
...
equLabel: EQU label+6
...
```

12.3.41 A2335: Exported Absolute SET Label is not Supported

Type: Error

Description: A label specified in front of a SET directive was specified previously in an XDEF directive. This is not allowed.

Example:

```
        XDEF setLabel
CstSec: SECTION
lab: DC.B 6
...
setLabel: SET $77AA
...
```

Tip: SET labels can be defined in a special file which can be included in each assembly file where the labels are referenced.

Example: File *const.inc*

```
...
setLabel: SET $77AA
...
File Test.asm
        INCLUDE "const.inc"
CstSec: SECTION
lab: DC.B 6
...
```

12.3.42 A2336: Value too Big

Type: Warning

Description: The absolute expression specified as the initialization value for a block, defined using DCB, is too big. This message is generated when the initial value specified in a DCB.B directive cannot be coded on a byte. In this case, the value used to initialize the constant block will be truncated to a byte value.

Example:

```
constSec: SECTION
…
label1:  DCB.B   2, 312
…
```

In the previous example, the constant block is initialized with the value $38 (= 312 and $FF).

Tip: To avoid this warning, modify the initialization value to a byte value.

Example:

```
constSec: SECTION
…
label1:  DCB.B   2, 56
…
```

12.3.43 A2338: <Message String>

Type: Error

Description: The FAIL directive followed by a string has been detected in the source file.

This is the normal behavior for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:
```
cpChar: MACRO
    IFC "\1", ""
      FAIL "A char must be specified as first parameter"
      MEXIT
    ELSE
      LDD \1
    ENDIF

    IFC "\2", ""
      FAIL 600  ; Warning
    ELSE
      STD \2
    ENDIF
  ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

12.3.44 A2341: Relocatable Section not Allowed: Absolute File is Currently Directly Generated

Type: Error

Description: A relocatable section has been detected while the assembler tries to generate an absolute file. This is not allowed.

Example:

```
DataSec: SECTION
             data1:   DS.W 1
                      ORG $800
             entry:
                      LDX #data1
```

Tips:

- When generating an absolute file, the application should be encoded in a single assembly unit and should not contain a relocatable symbol.

- To avoid this message, define sections as absolute and remove all XREF directives from the source file.

Example:

```
                      ORG $B00
             data1:   DS.W 1
                      ORG $800
             entry:
                      LDX #data1
```

12.3.45 A12001: Illegal Addressing Mode

Type: Error

Description: An illegal addressing mode has been detected in an instruction. This can be generated when an incorrect encoding is used for an addressing mode.

Example:

```
LDD [D X]
LDD [D, X
ANDCC $FA
```

Tip: Use a valid notation for the addressing mode encoding.

Example:

```
LDD [D, X]
ANDCC #$FA
```

12.3.46 A12002: Complex Relocatable Expression not Supported

Type: Error

Description: A complex relocatable expression has been detected. The expression is detected when it contains:

- An operation between labels located in two different sections

- A multiplication, division, or modulo operation between two labels

- The addition of two labels located in the same section

Example:

```
DataSec1: SECTION
DataLbl1: DS.B 10
DataSec2: SECTION
DataLbl2: DS.W 15
offset: EQU DataLbl2 – DataLbl1
```

Tip: The macro assembler does not support complex relocatable expressions. The corresponding expression must be evaluated at execution time.

Example:

```
DataSec1: SECTION
DataLbl1: DS.B 10
DataSec2: SECTION
DataLbl2: DS.W 15
Offset:   DS.W 1
…
CodeSec: SECTION
  …
evalOffset:
  LDD  #DataLbl2
  SUBD #DataLbl1
  STD  Offset
```

12.3.47 A12003: Value is Truncated to One Byte

Type: Warning

Description: A word operand is specified in an assembly instruction expecting a byte operand. This warning may be generated in the following cases:

- A symbol located in a section, which is accessed using the extended addressing mode, is specified as an operand in an instruction expecting a direct operand.

- An external symbol imported using XREF is specified as an operand in an instruction expecting a direct operand.

- The mask specified in BCLR, BSET, BRCLR, or BRSET is bigger than 0xFF.

Example:

```
                XREF extData
        dataSec: SECTION
        data:    DS.B 1
        data2:   DS.B 1
        destination: DS.W 1
        codeSec: SECTION
                MOVB    #data, destination
                MOVB    #data, destination
                MOVB    #extData, destination
                BCLR    data, #$54F
```

Tip: According to the reason why the warning was generated, the warning can be avoided as follows:

- Specify the force operator .B at the end of the operand or < in front of the operand.

- Use XREF.B to import the symbol.

Example:

```
                XREF.B extData
        dataSec: SECTION
        data:    DS.B 1
        data2:   DS.B 1
        destination: DS.W 1
        codeSec: SECTION
                MOVB    #data.B, destination
                MOVB    #extData, destination
                BCLR    data, #$4F
```

MCUez HC12 Assembler   User's Manual

MOTOROLA   Assembler Messages   287

12.3.48 A12005: Value Must Be Between 1 and 8

Type:	Error
Description:	The expression specified in a pre-increment, post-increment, pre-decrement, or post-decrement addressing mode is out of the range [1...8].
Example:	

```
        STX   10, SP+
```

Tip:	According to the HC12 addressing mode notation, the increment or decrement factor must be bigger than 0 and smaller than 9.

12.3.49 A12007: Comma Expected

Type:	Error
Description:	A comma character (,) is missing between two instructions or directive operands.
Example:	

```
        DataSec: SECTION SHORT
        Data:    DS.B  1
        CodeSec: SECTION
          MOVB   #$55 data
```

Tip:	Use the comma character as a separator between instruction operands.
Example:	

```
        DataSec: SECTION SHORT
        Data:    DS.B  1
        CodeSec: SECTION
          MOVB   #$55, data
```

12.3.50 A12008: Relative Branch with Illegal Target

Type: Error

Description: The offset specified in a PC relative addressing mode is a
complex relocatable expression.

Example:

```
DataSec: SECTION
Data:     DS.B  1
Code1Sec: SECTION
Entry1:
  NOP
  LDD #$6000
  STD Data
CodeSec: SECTION
  LDD    Data
  CPD    #$6000
  BNE    Entry1 *3
```

12.3.51 A12009: Illegal Expression

Type: Error

Description: An illegal expression is specified in a PC relative addressing mode. The illegal expression may be generated in two cases:

- A complex expression is specified when a PC relative expression is expected.

- A left or right parenthesis is missing in the expression.

Example:

```
CodeSec1: SECTION
Entry1:
  NOP
CodeSec2: SECTION
Entry2:
  NOP
  BRA #200
```

Tip: Change the expression to a valid expression.

Example:

```
CodeSec1: SECTION
Entry1:
  NOP
CodeSec2: SECTION
Entry2:
  NOP
  BRA Entry2

  BRA (Entry2 + 1)
```

12.3.52 A12010: Register Expected

Type: Error

Description: A register mnemonic is missing in a post-increment,
               post-decrement, pre-increment, or pre-decrement addressing
               mode.

Example:

```
        LDD   1, -ssp
```

Tip: Specify a register mnemonic at the specified position.

Example:

```
        LDD   1, -sp
```

12.3.53 A12011: Size Specification Expected

Type: Error

Description: An invalid size specification character is detected after a symbol name in an expression.

Valid size specification characters are:

`.B` for direct addressing mode

`.W` for extended addressing mode

Example:
```
DataSec: SECTION
...
label3 EQU label1.H +5

Insert a valid size specification character.

DataSec: SECTION
...
label3 EQU label1.B +5
```

12.3.54 A12102: Page Value Expected

Type: Error

Description: A page number is missing in a CALL instruction.

Example:

```
DataSec: SECTION
data: DS.L 2
FarCodeSec: SECTION
FarFunction:
    LDD   #45
    STD   data
CodeSec:    SECTION
…
  CALL  FarFunction
```

Tip: Add the missing page operand to the CALL instruction.

Example:

```
DataSec: SECTION
data: DS.L 2
FarCodeSec: SECTION
FarFunction:
    LDD   #45
    STD   data
CodeSec:    SECTION
…
CALL  FarFunction, PAGE(FarFunction)
```

12.3.55 A12103: Operand not Allowed

Type: Error

Description: The operand specified in an assembly instruction is not valid
                for this instruction.

Example:

```
DataSec: SECTION
data DS.B 20
…
CodeSec: SECTION
LEAX #data
```

Tip: Check the *CPU12 Reference Manual*, Motorola document
                order number CPU12RM/AD, and *CPU12 Reference Guide*,
                document order number CPU12RG/D, and ensure that the
                source code contains only valid instructions and addressing
                mode combinations.

Example:

```
DataSec: SECTION
data DS.B 20
…
CodeSec: SECTION
LDX #data
```

12.3.56 A12104: Immediate Value Expected

Type: Error

Description: The immediate addressing mode is expected at that position. Usually, this error message is generated when the mask specified in a BRCLR or BRSET instruction is not preceded by the immediate character (#).

Example:

```
maskValue: EQU $40
        BSCT
var:    DS.B 1
CodeSec: SECTION
entry:
        LDD   #4567
        BRCLR var, maskValue, endCode
        …
endCode:
        END
```

Tip: Insert the immediate character (#) at the requested position to change to the immediate addressing mode.

Example:

```
maskValue: EQU $40
        BSCT
var:    DS.B 1
CodeSec: SECTION
entry:
        LDD   #4567
        BRCLR var, #maskValue, endCode
        …
endCode:
        END
```

12.3.57 A12105: Immediate Address Mode not Allowed

Type: Error

Description: The immediate addressing mode is not allowed at that position. Usually, this message is generated when the first operand specified in a BCLR, BSET, BRCLR, or BRSET instruction is preceded by the immediate character (#).

Example:

```
maskValue: EQU $40
        BSCT
var:    DS.B 1
CodeSec: SECTION
entry:
        LDD   #4567
        BRCLR #var, #maskValue, endCode
        …
endCode:
        END
```

Tip: Remove the unexpected (#) character.

Example:

```
maskValue: EQU $40
        BSCT
var:    DS.B 1
CodeSec: SECTION
entry:
        LDD   #4567
        BRCLR var, #maskValue, endCode
        …
endCode:
        END
```

12.3.58 A12107: Illegal Size Specification for HC12 Instruction

Type: Error

Description: A size operator follows an HC12 instruction. Size operators are coded as a period character followed by a single character.

Example:

```
CodeSec: SECTION
   …
   ADDD.W #$0076
```

Tip: Remove the size specification following the HC12 instruction.

Example:

```
CodeSec: SECTION
   …
   ADDD #$0076
```

12.3.59 A12109: Illegal Character at the End of Line

Type: Error

Description: An invalid character or sequence of characters is detected at the end of an instruction. This message can be generated when:

- A comment specified after the instruction does not start with a comment character (;).

- An additional operand is specified in the instruction.

Example:

```
DataSec: SECTION
var:     DS.B 1
CodeSec: SECTION
  LDAA var Load A with the value of 'var'
  …
  LDAA var, #$44
```

Tips:

- Remove the invalid character or sequence of characters from the line.

- Insert the start of comment character at the beginning of the comment.

- Remove the extra operand.

Example:

```
DataSec: SECTION
var:     DS.B 1
CodeSec: SECTION
  LDAA var  ;Load  A  with  the  value  of
'var'
  …
  LDAA var
```

12.3.60 A12110: No Operand Expected

Type: Error

Description: An operand has been detected after an instruction that does not expect an operand.

Example:

```
CodeSec: SECTION
  PSHX toto
```

Tip: Remove the unexpected operand or put a semi-colon (;) in front of it to redefine it as a comment.

Example:

```
CodeSec: SECTION
  PSHX ;toto
```

12.3.61 A12201: Lexical Error in First or Second Field

Type: Error

Description: An incorrect assembly line is detected. This message may be
                generated when:

- An assembly instruction or directive starts on column 1.

- An assembly label on column 1 is not delimited by a colon
  character.

- An invalid identifier has been detected in the assembly line
  label or instruction. Characters allowed as the first character
  in an identifier are:
  A...Z, a...z, _, .
  Characters allowed as the first character in a label,
  instruction, or directive name are:
  A...Z, a...z, 0...9, _, .

Example:

```
CodeSec: SECTION
   …
LDD  #$320
   …
@label:
   …
4label:
```

Tips: Depending on why the message was generated, these actions
                can be taken:

- Insert at least one space in front of the directive or
  instruction.

- Insert a semicolon at the end of the label name.

- Change the label, directive, or instruction name to a valid
  identifier.

Example:

```
CodeSec: SECTION
   …
   LDD  #$320
   …
_label:
   …
_4label:
```

12.3.62 A12202: Not an HC12 Instruction or Directive

Type: Error

Description: The identifier detected in an assembly line instruction is not an assembly directive, a valid HC12 instruction, or a user-defined macro.

Example:

```
CodeSec: SECTION
   …
   LDHX #$5510
```

Tip: Change the identifier to an assembly directive, an HC12 instruction, or the name of a user-defined macro.

Example:

```
CodeSec: SECTION
   …
   LDX #$5510
```

12.3.63 A12203: Reserved Identifiers not Allowed as Instruction or Directive

Type: Error

Description: The identifier detected in an assembly line instruction is a reserved identifier. Reserved identifiers are:

• Mnemonics associated with target processor registers are A, B, CCR, D, PC, SP, TEMP2, TEMP3, X, Y.

• Mnemonics associated with a special target processor operator are PAGE.

Example:

```
CodeSec: SECTION
label:  X
```

Tip: Change the identifier name.

Example:

```
CodeSec: SECTION
label:  LDAA X
```

12.3.64 A12401: Value Out of Range –128...127

Type: Error

Description: The offset between the current PC and the label specified as the
PC relative address is not in the range of a signed byte (smaller
than –128 or bigger than 127). An 8-bit signed PC relative
offset is expected in the following instructions:

- Branch instructions:
  BCC, BCS, BEQ, BGE, BGT, BHI, BHS, BLE, BLO, BLS,
  BLT, BMI, BNE, BPL, BRA, BRN, BSR, BVC, BVS.

- Third operand in BRCLR and BRSET instructions

Example for branch instruction:
```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION

         …
         LDD  var1
         BNE  label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Tip: If one of the branch instructions has been used, use the
corresponding long-branch instruction.

Example:
```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION

         …
         LDD  var1
         LBNE label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Example for BRCLR instruction:

```
DataSec: SECTION
var1:    DS.W 100
CodeSec: SECTION

         …
         LDX  #var1
         BRCLR  3, X, #$05, label
dummyBl: DCB.B 200, $A7
label    STD var2
```

Tip: If a BRSET or BRCLR has been used, replace the BRCLR
                instruction with this code sequence:

```
LDAB  <first operand in the BRCLR>
ANDB  <second operand in BRCLR>
LBEQ  <third operand in BRCLR>
```

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION

         …
         LDX  #var1
         LDAB 3, X
         ANDB #$05
         LBEQ label
dummyBl: DCB.B 200, $A7
label    STD var2
```

12.3.65 A12402: Value Out of Range –32,768...32,767

Type:	Error
Description:	The offset between the current PC and label specified as the PC relative address is not in the range of a signed word (smaller than –32,768 or bigger than 32,767).

A 16-bit signed PC relative offset is expected in these instructions:

- Long-branch instructions:
  LBCC, LBCS, LBEQ, LBGE, LBGT, LBHI, LBHS, LBLE, LBLO, LBLS, LBLT, LBMI, LBNE, LBPL, LBRA, LBRN, LBVC, LBVS.

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION

         …
         LDD  var1
         LBNE label
dummyBl: DCB.B 20000, $A7
         DCB.B 20000, $A7
label    STD var2
```

Tip: Replace the long-branch instruction with this code sequence:

```
    <Inverse branch instruction> label1
        JMP  label
label1:
```

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 2
CodeSec: SECTION

         …
         LDD  var1
         BEQ  label1
         JMP  label
label1:
dummyBl: DCB.B 20000, $A7
         DCB.B 20000, $A7
label    STD var2
```

12.3.66 A12403: Value Out of Range –256...255

Type: Error

Description: The offset between the current PC and label specified as the PC relative address is not in the range of a signed 9-bit value (smaller than –256 or bigger than 255). A 9-bit signed PC relative offset is expected in these instructions:

- Decrement-and-branch instructions: DBEQ, DBNE

- Increment-and-branch instructions: IBEQ, INE

- Test-and-branch instructions: TBEQ, TBNE

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 10
CodeSec: SECTION

         …
         LDX  #var2
label:   LDD  var1
         CLR  1, X+
dummyBl: DCB.B 260, $A7
         DBNE D, label
```

Tips: Replace the instruction with the following portion of code.

- For decrement and branch:

IFCC	Condition
DBNE D, label	SUBD #1 LBNE label
DBNE A, label	DECA LBNE label
DBNE B, label	DECB LBNE label
DBNE X, label	DEX LBNE label
DBNE Y, label	DEY LBNE label
DBNE S, label	DES LBNE label

- For increment and branch:

IFCC	Condition
IBNE D, label	ADDD #1 LBNE label
IBNE A, label	INCA LBNE label
IBNE B, label	INCB LBNE label
IBNE X, label	INX LBNE label
IBNE Y, label	INY LBNE label
IBNE S, label	INS LBNE label

- For test and branch:

IFCC	Condition
TBNE D, label	CPD #1 LBNE label
TBNE A, label	TSTA LBNE label
TBNE B, label	TSTB LBNE label
TBNE X, label	CPX #1 LBNE label
TBNE Y, label	CPY #1 LBNE label
TBNE S, label	CPS #1 LBNE label

Example:

```
DataSec: SECTION
var1:    DS.W 1
var2:    DS.W 10
CodeSec: SECTION

         …
         LDX  #var2
label:   LDD  var1
         CLR  1, X+
dummyBl: DCB.B 260, $A7
         SUBD #1
         LBNE label
```

12.3.67 A12405: PAGE with Initialized RAM not Supported

Type: Error

Description: The PAGE operator has been specified in a DC directive. This
               restriction applies only to the MCUez file format.

Example:

```
adrEntry: DC.W entry
codeSec: SECTION
entry:
          NOP
          NOP
cstSec:  SECTION
pgEntry   DC.B PAGE(entry)
```

Tip: The complete address from the entry label can be loaded using
               a DC.L directive. The only drawback is that four bytes are
               allocated to store the address instead of three bytes.

Example:

```
codeSec: SECTION
entry:
          NOP
          NOP
cstSec:  SECTION
pgEntry   DC.L entry
```

12.3.68 A12408: Code Size Per Section Is Limited to 32 Kbytes

Type: Error

Description: One of the code or data sections defined in the application is bigger than 32 K. This is a limitation in the assembly version.

Example:

```
cstSec:   SECTION
noptable: DCB.L 4000, $A7
          DCB.L 4000, $A7
          DCB.L 4000, $A7
          DCB.L 500, $A7
```

Tip: Split the section into sections less than 32 K. The order in which the sections are allocated can be specified in the linker PRM (parameter) file. Specify that both sections are to be allocated consecutively, one after the other.

Example of assembly file:

```
cstSec:   SECTION
noptbl:   DCB.L 4000, $A7
          DCB.L 4000, $A7
cstSec1:  SECTION
noptbl1:  DCB.L 4000, $A7
          DCB.L 500, $A7
```

Example of PRM file:

```
LINK
  test.abs

NAMES test.o END
SECTIONS
    MY_RAM = READ_WRITE 0x0051 TO
0x00BF;
    MY_ROM = READ_ONLY  0x8301 TO
0x8DFD;
    ROM_2  = READ_ONLY  0xC000 TO
0xC1FD;
PLACEMENT
    DEFAULT_ROM     INTO MY_ROM;
    DEFAULT_RAM     INTO MY_RAM;
    cstSec, cstSec1 INTO ROM_2;
END
INIT entry
STACKSIZE 0x60
```

12.3.69 A12409: In PC Relative Addressing Mode, References to Object Located in Another Section or File Only Allowed for IDX2 Addressing Mode

Type: Error

Description: A reference to an external symbol or a symbol defined in another section is detected in a 9-bit or 5-bit indexed PC relative addressing mode. This is not allowed.

Example:

```
dataSec:   SECTION
data:      DS.W 1
cstSec:    SECTION
label:     DC.W $33A5, $44BA
codeSec1:  SECTION
entry:
           MOVB  label, PCR, data
```

Tip: Merge the sections containing the symbol and instruction or change the instruction to an instruction supporting the 16-bit indexed PC relative addressing mode.

Example of merging sections:

```
dataSec:   SECTION
data:      DS.W 1
codeSec1:  SECTION
label:     DC.W $33A5, $44BA
entry:
           MOVB  label, PCR, data
```

Example of changing instruction:

```
dataSec:   SECTION
data:      DS.W 1
cstSec:    SECTION
label:     DC.W $33A5, $44BA
codeSec1:  SECTION
entry:
           LDD   label, PCR
           STD   data
```

12.3.70 A12411:Restriction: Label Specified in a DBNE, DBEQ, IBNE, IBEQ, TBNE, or TBEQ Instruction Should Be Defined in the Same Section They Are Used

Type: Error

Description: An external symbol or a symbol defined in another section has been detected in a DBNE, DBEQ, IBNE, IBEQ, TBNE, or TBEQ instruction.

This is not allowed in a relocatable section.

Example:

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
           NOP
codeSec1:  SECTION
entry:
           DBNE   A, label
```

Tip: Merge the sections containing the symbol and instruction or change the instruction to an instruction that supports the 16-bit indexed PC relative addressing mode.

Example of merging sections:

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
           NOP
entry:
           DBNE   A, label
```

Example of changing instruction:

```
dataSec:   SECTION
data:      DS.W 1
codeSec0:  SECTION
label:
           NOP
           NOP
codeSec1:  SECTION
entry:
           DECA
           BNE    label
```

Appendix A. MASM Compatibility

A.1 Content

A.2 Introduction

The MCUez HC12 assembler has been extended to ensure compatibility with the MASM assembler.

A.3 Comment Line

A line starting with an asterisk (*) character is considered to be a comment line.

A.4 Constants

For compatibility with MASM, these integer constant notations are supported:

- Decimal constants are a sequence of decimal digits (0–9) followed by d or D.

- Hexadecimal constants are a sequence of hexadecimal digits (0–9, a–f, A–F) followed by h or H.

- Octal constants are a sequence of octal digits (0–7) followed by o, O, q, or Q.

- Binary constants are a sequence of binary digits (0–1) followed by b or B

Example:

```
512d            ; decimal representation
512D            ; decimal representation
200h            ; hexadecimal representation
200H            ; hexadecimal representation
1000o           ; octal representation
1000O           ; octal representation
1000q           ; octal representation
1000Q           ; octal representation
1000000000b     ; binary representation
1000000000B     ; binary representation
```

A.5 Operators

For compatibility with the MASM assembler, the operator notation in **Table A-1** is supported.

Table A-1. Operators

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

A.6 Directives

Table A-2 lists directives supported by MCUez for compatibility with MASM.

Table A-2. Directives

Operator	Notation	Description
RMB	DS	Defines storage for a variable
ELSEC	ELSE	Alternate of conditional block
ENDC	ENDIF	End of conditional block
NOL	NOLIST	Specifies that all subsequent instructions must not be inserted in the listing file
TTL	TITLE	Defines the user-defined title for the assembler listing file
GLOBAL	XDEF	Makes a symbol public (visible from outside)
PUBLIC	XDEF	Makes a symbol public (visible from outside)
EXTERNAL	XREF	Imports reference to an external symbol
XREFB	XREF.B	Imports reference to an external symbol located on the direct page
SWITCH	—	Allows switching to a section that has been previously defined
ASCT	ASCT: SECTION	Creates a predefined section with name id ASCT
BSCT	BSCT: SECTION SHORT	Creates a predefined section with name id BSCT. Variables defined in this section are accessed using the direct addressing mode.
CSCT	CSCT: SECTION	Creates a predefined section with name id CSCT
DSCT	DSCT: SECTION	Creates a predefined section with name id DSCT
IDSCT	IDSCT: SECTION	Creates a predefined section with name id IDSCT
IPSCT	IPSCT: SECTION	Creates a predefined section with name id IPSCT
PSCT	PSCT: SECTION	Creates a predefined section with name id PSCT

For More Information On This Product,
Go to: www.freescale.com

User's Manual — MCUez HC12 Assembler

Appendix B. MCUasm Compatibility

B.1 Contents

B.2 Introduction

The macro HC12 assembler has been extended to ensure compatibility with the MCUasm assembler. MCUasm compatibilty mode can be activated by specifying the option -MCUasm.

B.3 Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even if they start on column one.

Example:

```
label:  NOP
```

B.4 Set Directive

When MCUasm compatibility mode is activated, relocatable expressions are allowed in a SET directive.

Example:

```
label: SET *
```

If MCUasm compatibility mode is not activated, the SET label refers to absolute expressions.

B.5 Obsolete Directives

Table B-1 lists directives that are not recognized if MCUasm compatibility mode is activated.

Table B-1. Obsolete Directives

Operator	Notation	Description
RMB	DS	Defines storage for a variable
NOL	NOLIST	All subsequent instructions will not be inserted in the listing file.
TTL	TITLE	Defines title for assembler listing file
GLOBAL	XDEF	Makes a symbol public (visible from outside)
PUBLIC	XDEF	Makes a symbol public (visible from outside)
EXTERNAL	XREF	Imports reference to an external symbol

Index

Symbols

A

Freescale Semiconductor, Inc.

F

G

H

I

L

MCUez HC12 Assembler   User's Manual

W

X

Need to know more? That's ez, too.

Technical support for MCUez development tools is available through your regional Motorola office or by contacting:

Motorola, Inc.

6501 William Cannon Drive West

MD:OE17

Austin, Texas 78735

Phone (800) 521-6274

Fax (602) 437-1858

CRC@CRC.email.sps.mot.com

Freescale Semiconductor, Inc...

Ⓜ MOTOROLA

Semiconductor Produ

NXP