XGATE Assembler Manual

Revised: 23 October 2015

freescale™

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

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

How to Contact Us

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

Table of Contents

I Using XGATE Assembler

II Appendices

Using XGATE Assembler

This document explains how to effectively use the XGATE Macro Assembler.

Highlights

The major features of the XGATE Assembler are:

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

Structure of this Document

This section has the following chapters:

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

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

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

- Global Configuration File Entries: Description of the sections and entries that can appear in the global configuration file - `mcutools.ini`
- Local Configuration File Entries: Description of the sections and entries that can appear in the local configuration file - `*project*.ini`

1

Working with Assembler

This chapter is primarily a tutorial for creating and managing XGATE assembly projects with the CodeWarrior Development Studio for S12(X) v5.x. In addition, there are instructions to utilize the Assembler and Smart Linker Build Tools in the CodeWarrior Development Studio for S12(X) v5.x for assembling and linking assembly projects.

This chapter covers the following topics:

- Programming Overview
- Managing Assembly Language Project
- Analysis of Groups and Files in Project Window
- Writing Assembly Source Files
- Analyzing Project Files
- Assembling Source Files
- Linking Application
- Directly Generating ABS File
- Using Assembler for Absolute Assembly

Programming Overview

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

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

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

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

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

The XGATE Assembler can be used as a transparent, integral part of the CodeWarrior Development Studio for S12(X) v5.x. This is the recommended way to get your project up and running in minimal time. Alternatively, the Assembler can also be configured and used as a standalone macro assembler as one of the Build Tool Utilities included with CodeWarrior such as a Linker, Compiler, ROM Burner, Simulator or Debugger, etc.

The typical configuration of an Assembler (or any of the other Build Tool Utilities) is its association with a Project Directory and an External Editor. The CodeWarrior Development Studio for S12(X) v5.x uses a project directory for storing the files it creates and coordinates the various Build Tools. The Assembler is but one of these tools that the CodeWarrior IDE coordinates. The tools used most frequently within CodeWarrior are its integrated Editor, Compiler, Assembler, Linker, the Simulator/Debugger, and Processor Expert. Most of these “Build Tools” are located in the *prog* subfolder of the CodeWarrior installation. The others are directly integrated into the CodeWarrior IDE.

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

Project Directory

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

In the process of designing a project, you can either start from scratch by designing your own source-code, configuration (`*.ini`), and various layout files for your project for use with standalone project-building tools. This was how embedded microprocessor projects were developed in the recent past. On the other hand, you can have the CodeWarrior IDE coordinate the Build Tools and transparently manage the entire project. This is recommended because it is far easier and faster than employing standalone tools. However, you can still utilize any of the separate Build Tools in the CodeWarrior Development Studio for S12(X) v5.x suite.

External Editor

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

Managing Assembly Language Project

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

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

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

Listing 1.1 Fibonacci series

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

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

Let’s create a project with CodeWarrior and analyze the assembly source and the Linker’s parameter files to calculate a Fibonacci series for a particular 16-bit microprocessor in the Freescale HCS12XA family - in this case, the MC9S12XA256.

Using New Project Wizard to Create Project

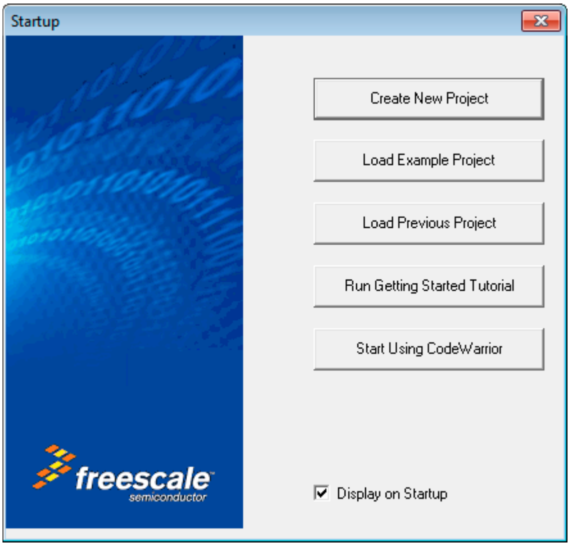
This section demonstrates using the CodeWarrior IDE Wizard to create a new project.

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

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

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

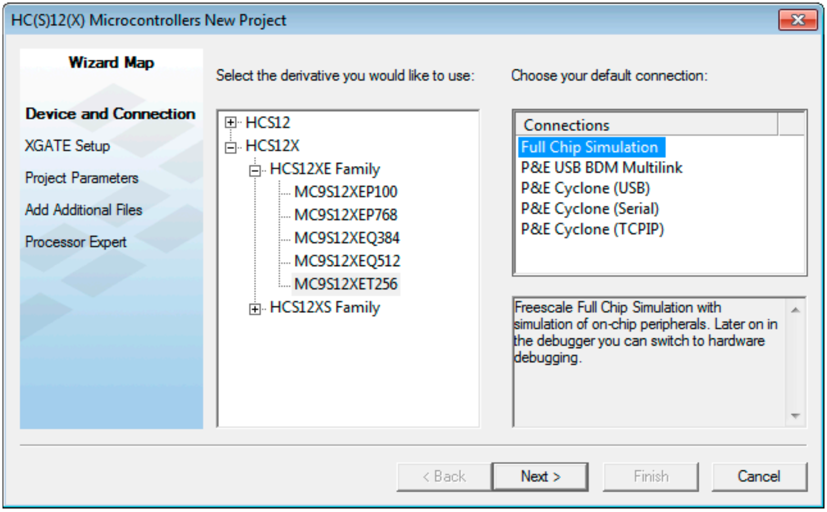
Figure 1.1 Startup Dialog Box



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

3. Select a derivative you would like to use. For example, select `HCS12X > HCS12XE Family > MC9S12XSET56`.

Figure 1.2 Device and Connection Page



Select the derivative you would like to use:

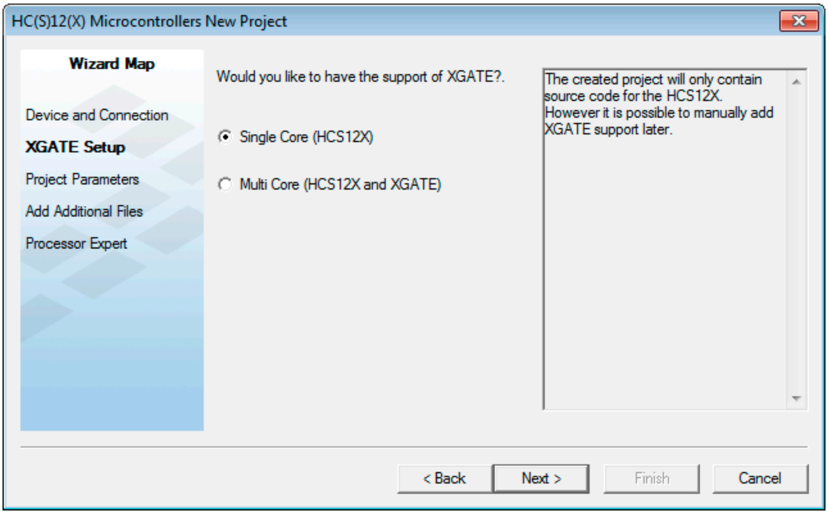
Choose your default connection:

Freescale Full Chip Simulation with simulation of on-chip peripherals. Later on in the debugger you can switch to hardware debugging.

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

   The **XGATE Setup** page (Figure 1.3) appears.

Figure 1.3 XGATE Setup Page



Would you like to have the support of XGATE?.

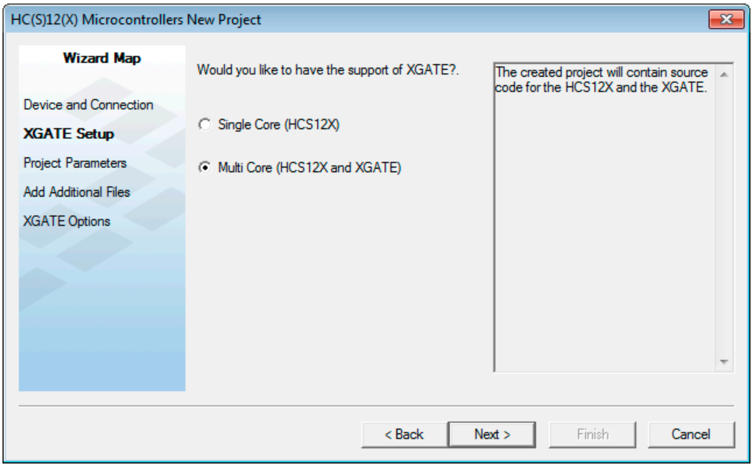
● Single Core (HCS12X)

○ Multi Core (HCS12X and XGATE)

The created project will only contain source code for the HCS12X. However it is possible to manually add XGATE support later.

6. Select the Multi Core (HCS12X and XGATE) option button to include support of XGATE (Figure 1.4).

Figure 1.4 XGATE Setup Page — Multi Core (HCS12X and XGATE)



HC(S)12(X) Microcontrollers New Project

Would you like to have the support of XGATE?.

○ Single Core (HCS12X)

● Multi Core (HCS12X and XGATE)

The created project will contain source code for the HCS12X and the XGATE.

7. Click the **Next** button.

   The **Project Parameters** page (Figure 1.5) appears.

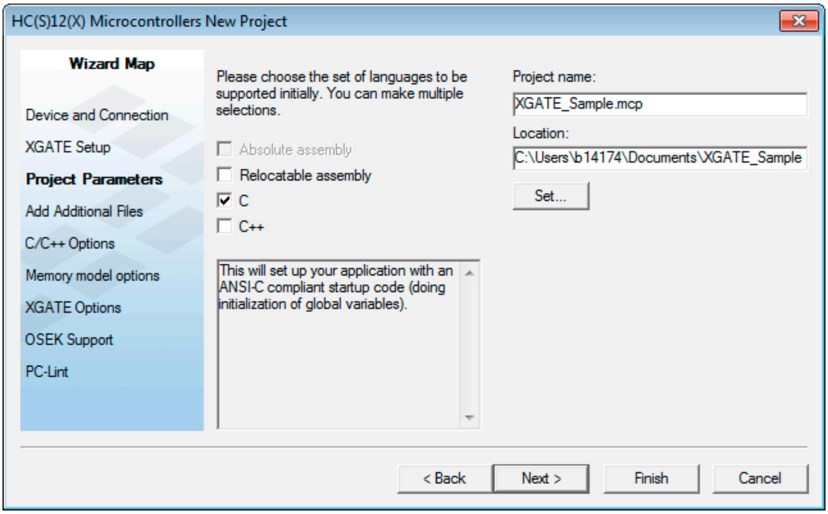
NOTE The page displayed may vary depending on the selected derivative and XGATE support.

8. Check the Relocatable Assembly checkbox and clear *C* and C++.
9. In the Project name text box, specify the name of the new project. For example, XGATE_Sample.
10. To specify a location other than the default location, enter the new path in the Location text box or click the **Set** button to browse the folder location.

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

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

Figure 1.5 Project Parameters Page



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

□ Absolute assembly
□ Relocatable assembly
☑ C
□ C++

This will set up your application with an ANSI-C compliant startup code (doing initialization of global variables).

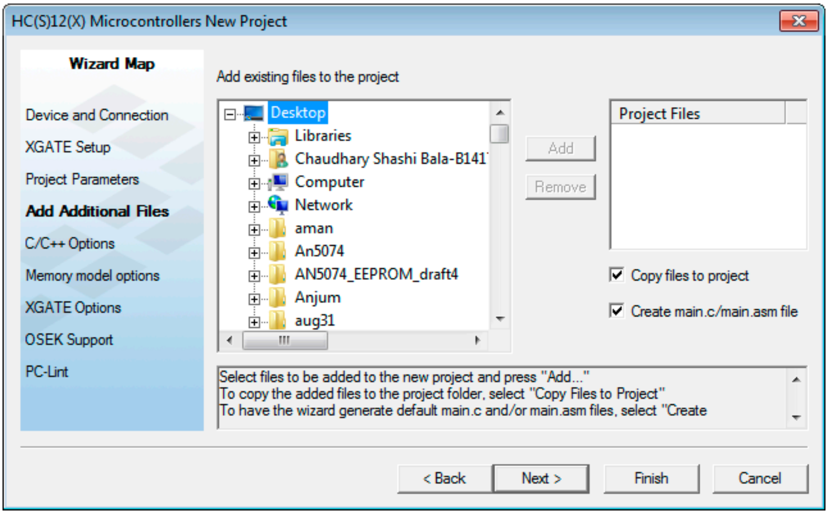
Project name: XGATE_Sample.mcp

Location: C:\Users\b14174\Documents\XGATE_Sample

11. Click the **Next** button.

The **Add Additional Files** page (Figure 1.6) appears.

Figure 1.6 Add Additional Files Page



Add existing files to the project

☑ Copy files to project
☑ Create main.c/main.asm file

Select files to be added to the new project and press "Add..."
To copy the added files to the project folder, select "Copy Files to Project"
To have the wizard generate default main.c and/or main.asm files, select "Create

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

The **C/C++ Options** page appears.

13. Select the startup code, memory model, and floating point format to be added to the project and click the **Next** button.

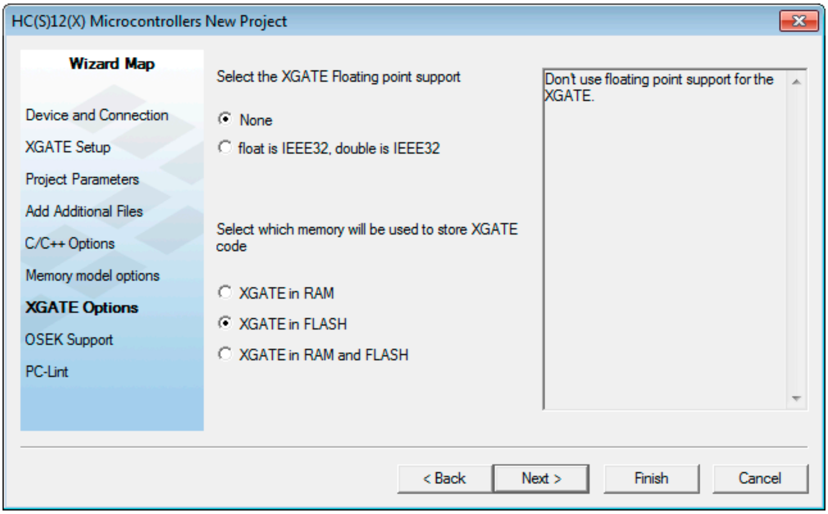
    The **Memory** model options page appears.

14. Select the required options or leave the settings as is and click the **Next** button.

    The **XGATE Options** page (Figure 1.7) appears.

15. Select the None option button for XGATE floating point format to support.
16. Select the XGATE in RAM option button to store XGATE code.

Figure 1.7 XGATE Options Page



Select the XGATE Floating point support

● None
○ float is IEEE32, double is IEEE32

Select which memory will be used to store XGATE code

○ XGATE in RAM
● XGATE in FLASH
○ XGATE in RAM and FLASH

Don't use floating point support for the XGATE.

17. Click the **Next** button.

    The **OSEK Support** page appears.

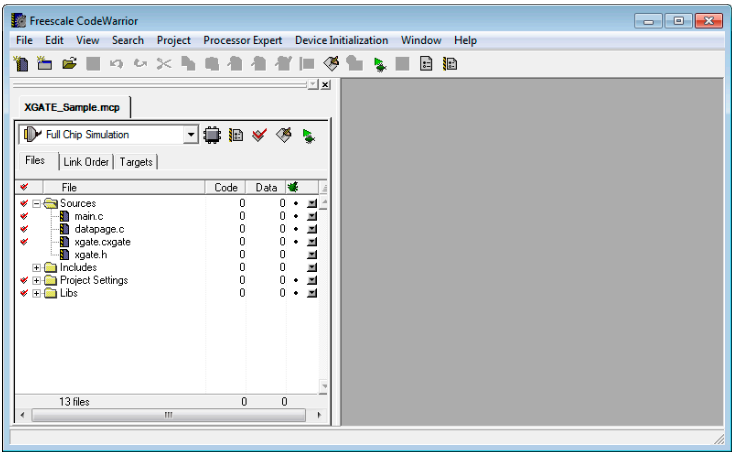
18. Select whether you want to add OSEK support to your project.
19. Click the **Next** button.

    The **PC-Lint** page appears.

20. Select whether you want to add PC-lint support to your project.
21. Click the **Finish** button.

    IDE creates a project according to your specifications and the project window (Figure 1.8) appears, docked at the left side of the main window.

Figure 1.8 CodeWarrior Project Window


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

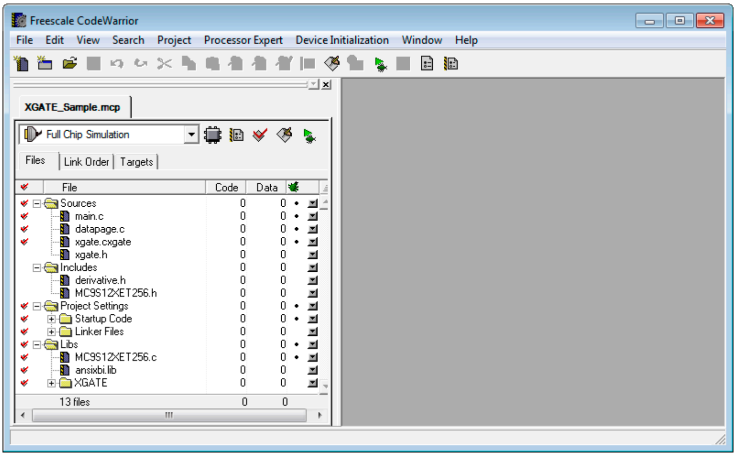
✓	File	Code	Data	
✓	Sources	0	0	•
✓	main.c	0	0	•
✓	datapage.c	0	0	•
✓	xgate.cxgate	0	0	•
	xgate.h	0	0	
	Includes	0	0	
✓	Project Settings	0	0	•
✓	Libs	0	0	•
	13 files	0	0	

Some files and folders are automatically generated. The root folder is the project directory that you selected in the first step.

NOTE You can (but do it later) safely close the CodeWarrior IDE at any time after this point, and your project will be automatically configured in its previously-saved status when you work on the project later. Using the New Project Wizard, an S12(X) XGATE project is set up quickly. You can add additional components to your project afterwards. Related files and folders are automatically generated in the root folder that was used in the project-naming process. This folder is referred to in this manual as the project directory.

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

Figure 1.9 Expanded View of Folders


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

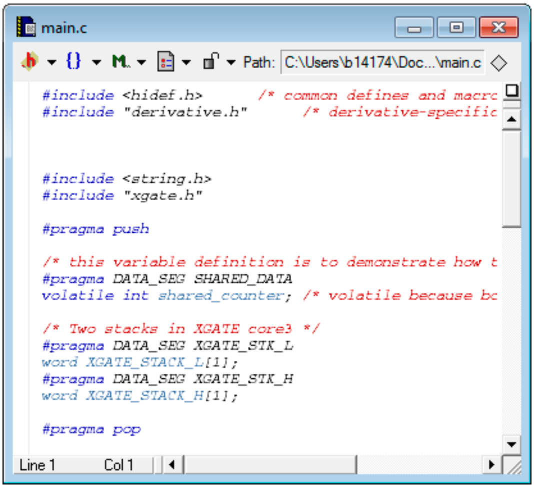
✓	File	Code	Data	
✓	Sources	0	0	•
✓	main.c	0	0	•
✓	datapage.c	0	0	•
✓	xgate.cxgate	0	0	•
	xgate.h	0	0	
	Includes	0	0	
	derivative.h	0	0	
	MC9S12XET256.h	0	0	
✓	Project Settings	0	0	•
✓	Startup Code	0	0	•
✓	Linker Files	0	0	
✓	Libs	0	0	•
✓	MC9S12XET256.c	0	0	•
✓	ansixbi.lib	0	0	
✓	XGATE	0	0	
	13 files	0	0	

NOTE You could use the Windows Explorer to examine the actual folders and files that CodeWarrior generated for your project and displays in the project window. Right-click on a file and select Open in Windows Explorer.

NOTE The major file for any CodeWarrior project is its `<project_name>.mcp` file. This is the file to reopen your project.

22. Double-click the `main.c` file in the **Sources** group. The CodeWarrior editor opens the `main.c` file (Figure 1.10).

Figure 1.10 `main.asm` File in the Editor Window



```
#include <hidef.h>      /* common defines and macrc
#include "derivative.h"      /* derivative-specific



#include <string.h>
#include "xgate.h"

#pragma push

/* this variable definition is to demonstrate how t
#pragma DATA_SEG SHARED_DATA
volatile int shared_counter; /* volatile because bc

/* Two stacks in XGATE core3 */
#pragma DATA_SEG XGATE_STK_L
word XGATE_STACK_L[1];
#pragma DATA_SEG XGATE_STK_H
word XGATE_STACK_H[1];

#pragma pop
```

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

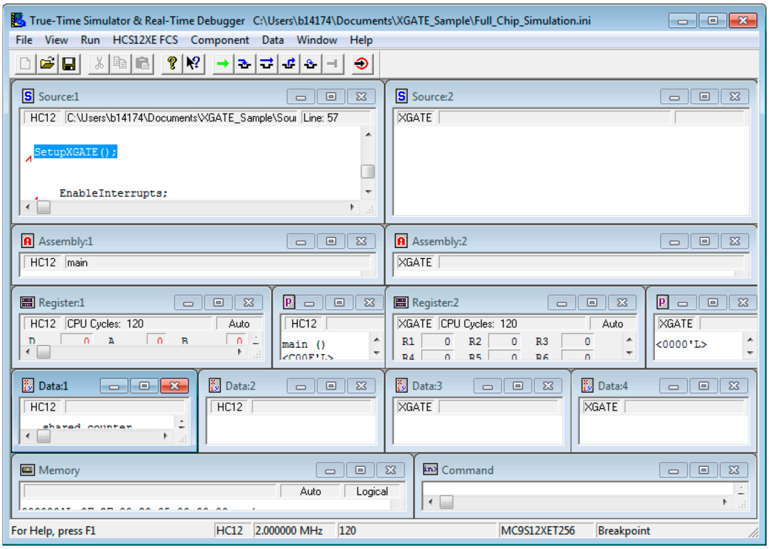
As a precaution, you can determine if the project is configured correctly and the source code is free of syntactical errors. It is not necessary that you do so, but you should make (build) the default project that the CodeWarrior Assembler just created.

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

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

    The**True-Time Simulator & Real-Time Debugger** window opens (Figure 1.11).

Figure 1.11 True-Time Simulator & Real-Time Debugger Window


True-Time Simulator & Real-Time Debugger C:\Users\b14174\Documents\XGATE_Sample\Full_Chip_Simulation.ini
File View Run HCS12XE FCS Component Data Window Help
Source:1
HC12 C:\Users\b14174\Documents\XGATE_Sample\Sou Line: 57
Source:2
XGATE
Assembly:1
HC12 main
Assembly:2
XGATE
Register:1
HC12 CPU Cycles: 120 Auto
Register:2
XGATE CPU Cycles: 120 Auto
R1 0 R2 0 R3 0
HC12
XGATE
Data:1
HC12
Data:2
HC12
Data:3
XGATE
Data:4
XGATE
Memory
Auto Logical
Command
For Help, press F1 HC12 2.000000 MHz 120 MC9S12XET256 Breakpoint

```
SetupXGATE();

  EnableInterrupts;
```

```
main ()
```

```
<0000'L>
```

Analysis of Groups and Files in Project Window

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

CodeWarrior Groups

These groups and their usual functions are:

- Sources

  This group contains the assembly source code files.

- Includes

  This project has an include file for the particular CPU derivative. In this case, the group contains the `mc9s12xa256.inc` file for the MC9S12XA256 derivative.

- Project Settings

This groups includes the Linker Files group.

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

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

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

Writing Assembly Source Files

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

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

he usual procedure for writing an assembly source-code file is to use the editor that is integrated into the CodeWarrior Development Studio for S12(X) v5.x. You can begin a new file by pressing the *New Text File* icon on the Toolbar to open a new file, write your assembly-source code, and later save it with a `*.asm` file extension using the *Save* icon on the Toolbar to name and store it wherever you want it placed - usually in the *Sources* folder.

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

Analyzing Project Files

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

Listing 1.2 main.asm file

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

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

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

            INCLUDE "xgate.inc"
            XREF XGATE_Counter         ; shared data with XGATE.

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

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

SetupXGATE:
 ifndef __RUN_XGATE_OUT_OF_FLASH ; do we need to copy the XGATE code
                                 ; into the RAM?
            ; copy the XGATE program code to the RAM
           LDY   #GLOBAL     (__SEG_START_XGATE_CODE)  ;we use GPAGE
                                                       ; for the
                                                       ; src (to flash)
            LDAB  #GLOBAL_PAGE(__SEG_START_XGATE_CODE)
            STAB  GPAGE_ADR
           LDX   #     __SEG_RELOCATE_TO_XGATE_CODE    ; and RPAGE to
                                                           ;the RAM
            LDAB  #PAGE(__SEG_RELOCATE_TO_XGATE_CODE)
            STAB  RPAGE_ADR

            LDD   #__SEG_SIZE_XGATE_CODE
            PSHD
CopyLoop:   GLDAA 1,Y+            ; load a single code byte from flash
            STAA  1,X+             ; store it
            DECW  0,SP
            BNE   CopyLoop
            PULD
 endif ; __RUN_XGATE_OUT_OF_FLASH

            ; init variables shared with XGATE
            CLRW XGATE_Counter    ; init counter for the XGATE

            ; initialize the XGATE vector block and
            ; set the XGVBR register to its start address
            LDX   #GLOBAL(XGATE_VectorTable)
            STX   $386  ; XGVBR

        ; switch software trigger 0 interrupt to XGATE
       MOVB #(SOFTWARETRIGGER0_VEC & $F0), INT_CFADDR
      MOVB #$80|$01, INT_CFDATA0 + ((SOFTWARETRIGGER0_VEC & $0F) >> 1)

       ; enable XGATE mode and interrupts
       MOVW #$FBC1, XGMCTL   ; XGE | XGFRZ | XGIE

       ; force execution of software trigger 0 handler
       MOVW #$0101, XGSWT


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


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

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

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

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

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

Assembling Source Files

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

Assembling with CodeWarrior

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

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

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

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

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

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

- Project.map

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

- Project.abs.s19

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

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

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

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

Assembling with Standalone Assembler

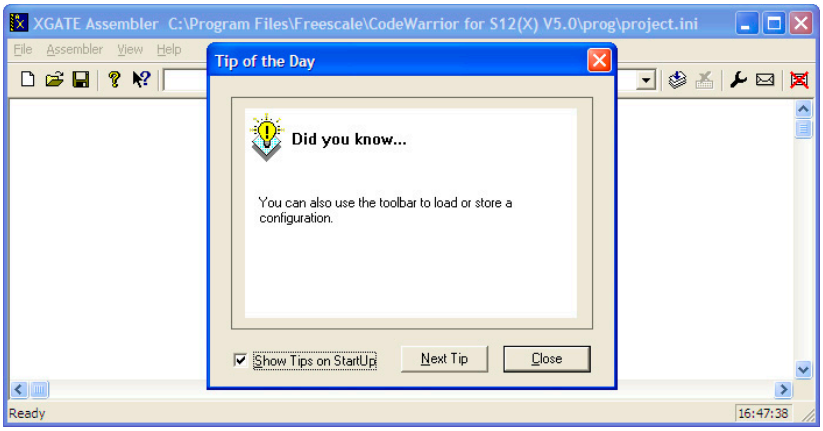
It is also possible to use the XGATE Assembler as a standalone assembler. If you already have an assembled source file and prefer not to use the Assembler but want to use the Linker, you can skip this section and proceed to “Linking Application” on page 48.

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

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

Start the Assembler by opening the `axgate.exe` file located in `<CodeWarrior Install dir>\Prog` folder. The Assembler opens (Figure 1.12).

Figure 1.12 XGATE Assembler Default Configuration Window


XGATE Assembler C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\prog\project.ini
File
Assembler
View
Help
Tip of the Day
Did you know...

You can also use the toolbar to load or store a configuration.

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

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

Configuring Assembler

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

- Global

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

Listing 1.3 Typical locations for a global configuration file

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

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

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

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

- Local

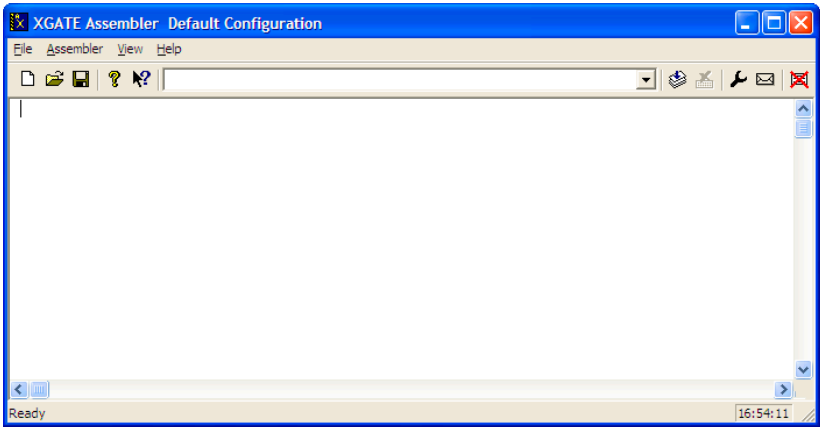
  This file could be used by any Build Tool for a particular project. For information about entries for the local configuration file, see Local Configuration File Entries in the Appendices.

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

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

The **XGATE Assembler Default Configuration** window appears (Figure 1.13)

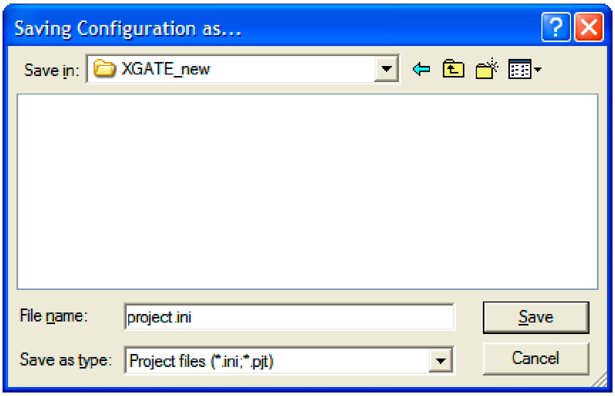
Figure 1.13 XGATE Assembler Default Configuration Window



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

   A **Save Configuration as** dialog box appears. Navigate to the folder of your choice and create and name a folder and filename for the configuration file (Figure 1.14).

Figure 1.14 Saving Configuration as... Dialog Box



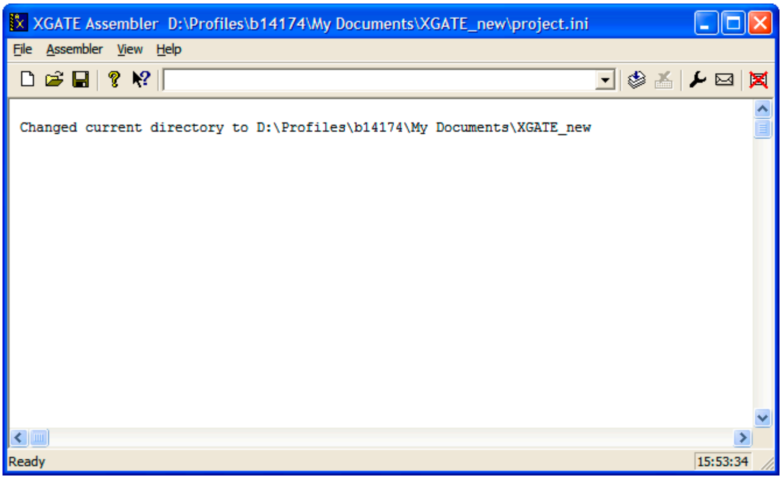
File name: project.ini

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

4. Click the **Save** button.

   The current directory for the assembler changes to your project directory (Figure 1.15).

Figure 1.15 Assembler Changed Current Directory to Project Directory


XGATE Assembler D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
Assembler
View
Help
Ready
15:53:34

```
Changed current directory to D:\Profiles\b14174\My Documents\XGATE_new
```

If you were to examine the project directory with Windows Explorer at this point, it would only contain the <*project_name*>*.ini* configuration file that you just created. If you further examined the contents of the project's configuration file, you would notice that it now contains the [AXGATE_Assembler] portion of the `project.ini` file in the `prog` folder where the Build Tools are located. Any options added to or deleted from your project by any Build Tool would be placed into or deleted from this configuration file in the appropriate section for each Build Tool.

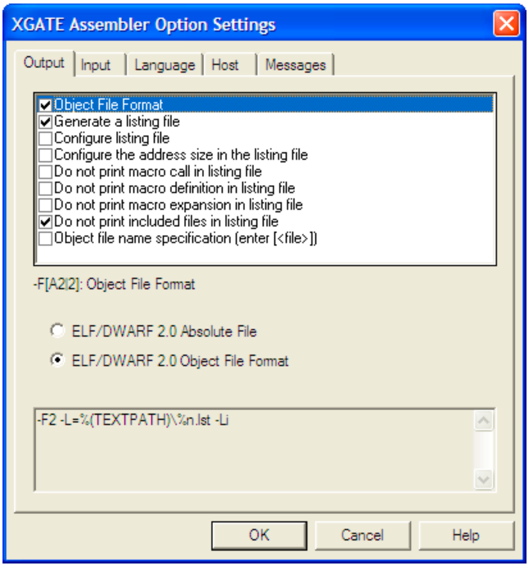
If you want some additional options to be applied to all projects, you can take care of that later by modifying the `project.ini` file in the `prog` folder.

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

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

   The Assembler displays the **XGATE Assembler Option Settings** dialog box (Figure 1.16).

Figure 1.16 XGATE Assembler Option Settings Dialog Box



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

-F[A2|2]: Object File Format

○ ELF/DWARF 2.0 Absolute File
● ELF/DWARF 2.0 Object File Format

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

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

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

4. Click the **OK** button to close the **XGATE Assembler Option Settings** dialog box.
5. Save the changes to the configuration by:
   - selecting **File > Save Configuration (Ctrl + S)** or
   - clicking the **Save** button on the toolbar.

Input Files

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

Now there are two files in the project:

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

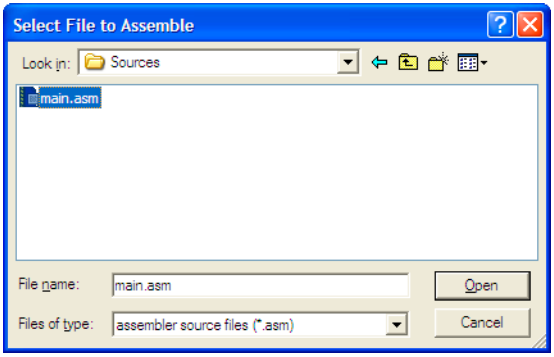
Assembling Assembly Source-Code Files

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

1. Select **File > Assemble**.

The **Select File to Assemble** dialog box appears (Figure 1.17).

Figure 1.17 Select File to Assemble Dialog Box



Look in: Sources

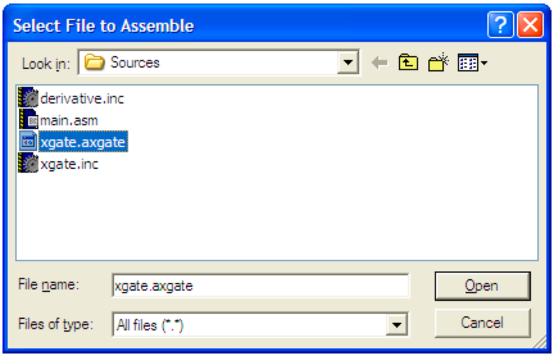
main.asm

File name: main.asm

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

2. Browse to the *`Sources`* folder in the project directory and select All files from the File of type list box.
3. Select the *`xgate.axgate`* file (Figure 1.19).

Figure 1.18 Select `xgate.axgate` File



Look in: Sources

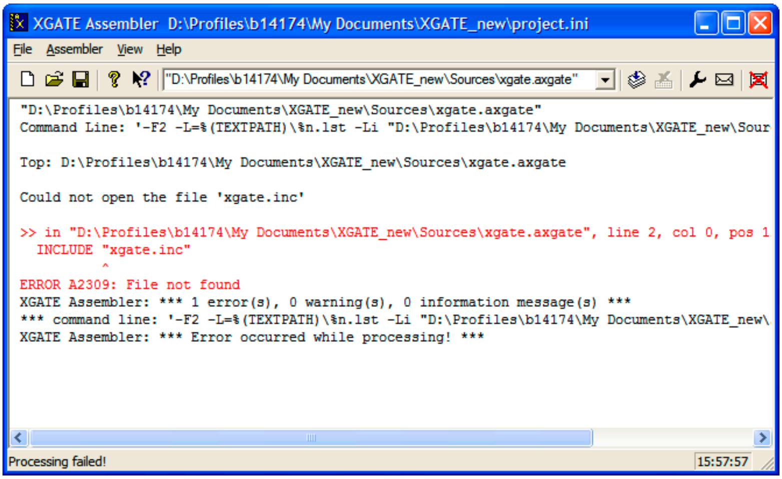
derivative.inc
main.asm
xgate.axgate
xgate.inc

File name: xgate.axgate

Files of type: All files (*.*)

4. Click the **Open** button and the `xgate.axgate` file should start assembling (Figure 1.19).

Figure 1.19 Results of assembling the xgate.axgate file...

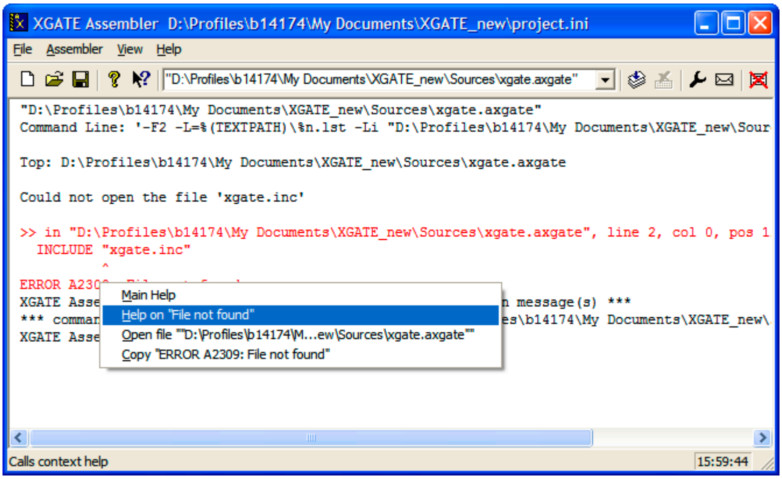

XGATE Assembler D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
Assembler
View
Help
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Processing failed!
15:57:57

```
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\XGATE_new\Sour

Top: D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate

Could not open the file 'xgate.inc'

>> in "D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate", line 2, col 0, pos 1
  INCLUDE "xgate.inc"
          ^
ERROR A2309: File not found
XGATE Assembler: *** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\XGATE_new\
XGATE Assembler: *** Error occurred while processing! ***
```

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

5. Right-click on the text about the error message, a context menu appears (Figure 1.20).

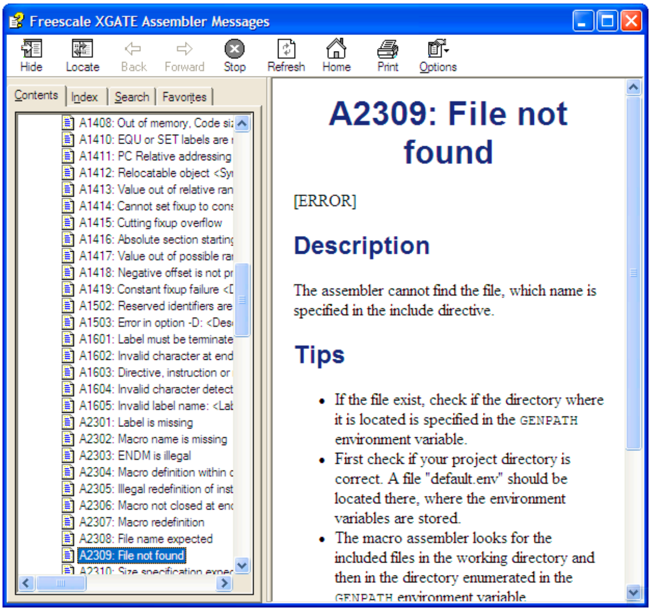
Figure 1.20 Context Menu


XGATE Assembler D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
Assembler
View
Help
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Main Help
Help on "File not found"
Open file ""D:\Profiles\b14174\M...ew\Sources\xgate.axgate""
Copy "ERROR A2309: File not found"
Calls context help
15:59:44

```
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\XGATE_new\Sour

Top: D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate

Could not open the file 'xgate.inc'

>> in "D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate", line 2, col 0, pos 1
  INCLUDE "xgate.inc"
          ^

ERROR A230
XGATE Asse                                                 n message(s) ***
*** comman                                                 es\b14174\My Documents\XGATE_new\
XGATE Asse
```

6. Select the *Help on "file not found"* option and help for the A2309 error message appears (Figure 1.21).

Figure 1.21 A2309 Error Message Help


Freescale XGATE Assembler Messages
Hide
Locate
Back
Forward
Stop
Refresh
Home
Print
Options
Contents
Index
Search
Favorites
A1408: Out of memory, Code siz
A1410: EQU or SET labels are
A1411: PC Relative addressing
A1412: Relocatable object <Sy
A1413: Value out of relative ran
A1414: Cannot set fixup to cons
A1415: Cutting fixup overflow
A1416: Absolute section startin
A1417: Value out of possible ra
A1418: Negative offset is not pr
A1419: Constant fixup failure <
A1502: Reserved identifiers are
A1503: Error in option -D: <Des
A1601: Label must be terminate
A1602: Invalid character at end
A1603: Directive, instruction or
A1604: Invalid character detect
A1605: Invalid label name: <Lab
A2301: Label is missing
A2302: Macro name is missing
A2303: ENDM is illegal
A2304: Macro definition within
A2305: Illegal redefinition of inst
A2306: Macro not closed at en
A2307: Macro redefinition
A2308: File name expected
A2309: File not found
A2310: Size specification expec

A2309: File not found

[ERROR]

Description

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

Tips

- If the file exist, check if the directory where it is located is specified in the `GENPATH` environment variable.
- First check if your project directory is correct. A file "default.env" should be located there, where the environment variables are stored.
- The macro assembler looks for the included files in the working directory and then in the directory enumerated in the `GENPATH` environment variable.

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

This implies that the `GENPATH` environment variable should specify the location of the `xgate.inc` include file.

NOTE If you read the `xgate.inc` file, you could have anticipated this on account of this statement on line 19: INCLUDE 'xgate.inc'.

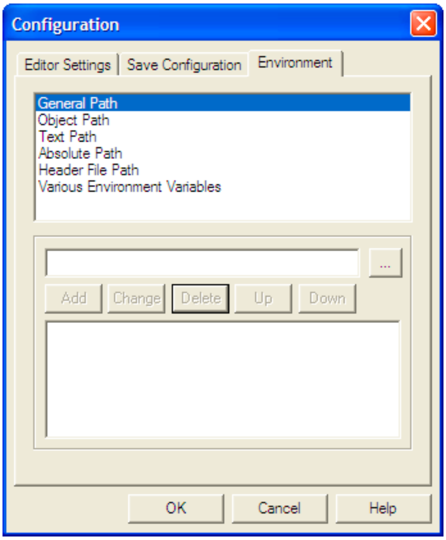
To fix this, perform these steps:

1. Select **File > Configuration**.

   The **Configuration** dialog box appears (Figure 1.22).

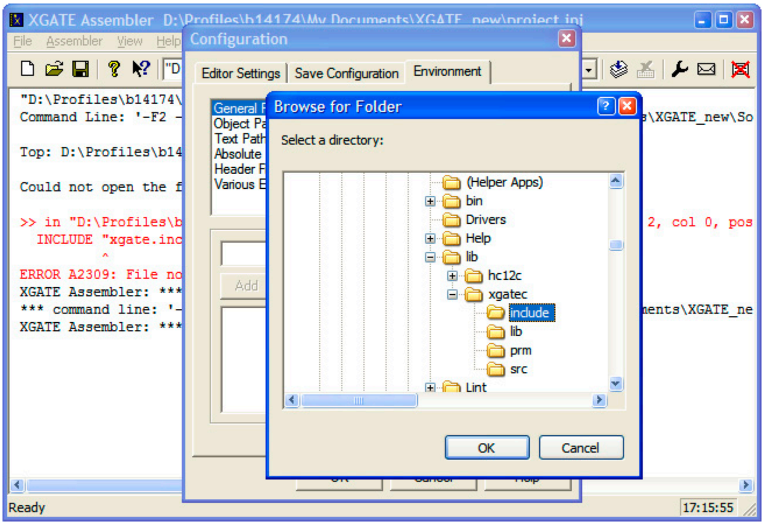
2. Select the **Environment** tab and then select **General Path**.

Figure 1.22 Configuration Dialog Box



3. Click the “**...**” button and navigate in the **Browse for Folder** dialog box for the folder that contains the missing file - the `include` subfolder in the CodeWarrior installation’s `lib` folder (Figure 1.23).

Figure 1.23 Browsing for Sources Folder

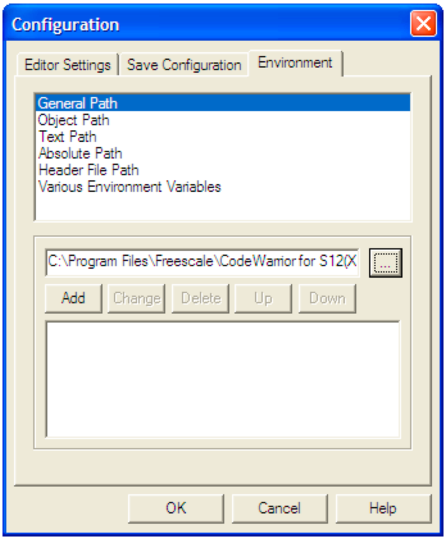


```
"D:\Profiles\b14174\                    s\XGATE_new\So
Command Line: '-F2 -

Top: D:\Profiles\b14

Could not open the f

>> in "D:\Profiles\b                    2, col 0, pos
  INCLUDE "xgate.inc
        ^
ERROR A2309: File no
XGATE Assembler: ***
*** command line: '-                    ments\XGATE_ne
XGATE Assembler: ***
```

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

   The **Configuration** dialog box is now active (Figure 1.24).

Figure 1.24 Adding GENPATH



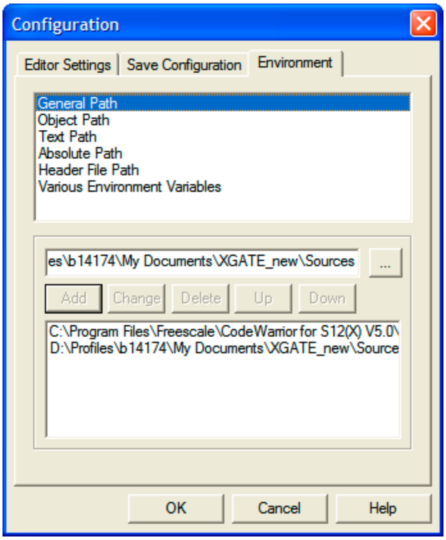
C:\Program Files\Freescale\CodeWarrior for S12(X

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

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

7. Similarly, add the `GENPATH` for the `xgate.inc` file. For this example, the path is `D:\Profiles\b14174\My Documents\XGATE_new\Sources` (Figure 1.26).

Figure 1.25 Adding GENPATH — xgate.inc



es\b14174\My Documents\XGATE_new\Sources

C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\
D:\Profiles\b14174\My Documents\XGATE_new\Source

8. Click the **OK** button to close the **Configuration** dialog box.

   An asterisk now appears in the Configuration title bar.

9. Click the **Save** button or select **File > Save Configuration** to save the changes to the configuration.

   The asterisk disappears.

NOTE If you do not save the configuration, the assembler will revert to the last-saved configuration the next time the project is reopened.

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

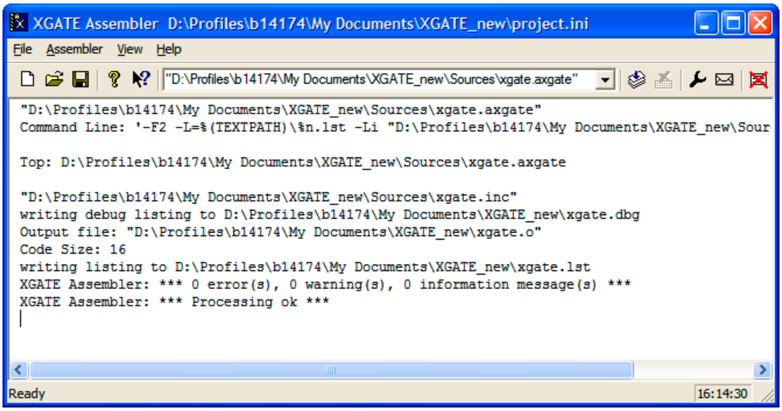
Now that you have supplied the path to the missing files, you can try again to assemble the xgate.axgate file.

1. Select **File > Assemble** and again navigate to the `Sources` folder (if it is not active).
2. Select the `xgate.axgate` file and click **Open**.

The Macro Assembler indicates successful assembly and indicates the Code Size in bytes. The message “`*** 0 error(s),`” indicates that the `main.asm` file assembled without errors. Save the configuration again.

NOTE Do not forget to save the configuration one additional time.

Figure 1.26 Successful Assembly - `xgate.o` Object File Created

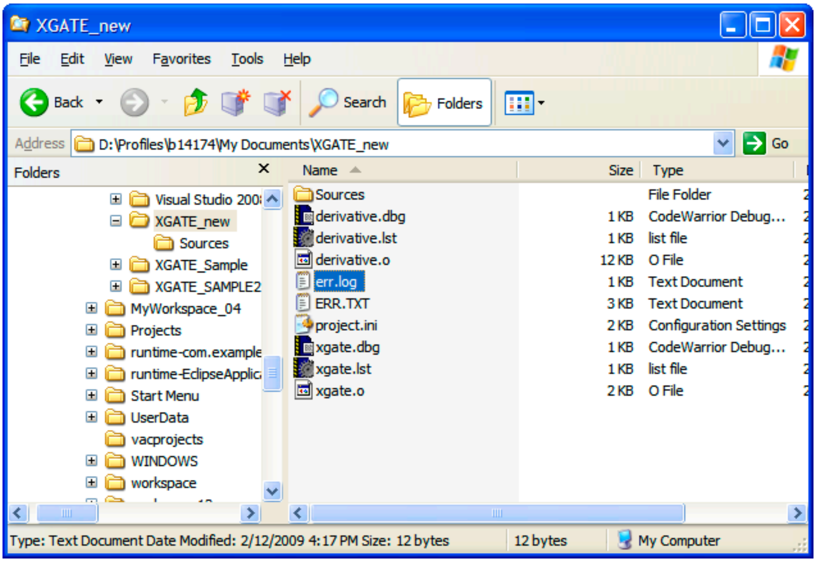

XGATE Assembler D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File Assembler View Help
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Ready
16:14:30

```
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Command Line: '-F2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\XGATE_new\Sour

Top: D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate

"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.inc"
writing debug listing to D:\Profiles\b14174\My Documents\XGATE_new\xgate.dbg
Output file: "D:\Profiles\b14174\My Documents\XGATE_new\xgate.o"
Code Size: 16
writing listing to D:\Profiles\b14174\My Documents\XGATE_new\xgate.lst
XGATE Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
XGATE Assembler: *** Processing ok ***
```

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

NOTE To assemble the main.asm in the `Sources` folder, locate and open the `ahc12.exe` in `<CodeWarrior Install dir>\prog` folder, load the `project.ini` file, and specify the `GENPATH` to the `xgate.o` file.

Figure 1.27 Project Directory After a Successful Assembly



Address: D:\Profiles\b14174\My Documents\XGATE_new

Name	Size	Type
Sources		File Folder
derivative.dbg	1 KB	CodeWarrior Debug...
derivative.lst	1 KB	list file
derivative.o	12 KB	O File
err.log	1 KB	Text Document
ERR.TXT	3 KB	Text Document
project.ini	2 KB	Configuration Settings
xgate.dbg	1 KB	CodeWarrior Debug...
xgate.lst	1 KB	list file
xgate.o	2 KB	O File

Type: Text Document Date Modified: 2/12/2009 4:17 PM Size: 12 bytes   12 bytes

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

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

Linking Application

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

Linking with CodeWarrior

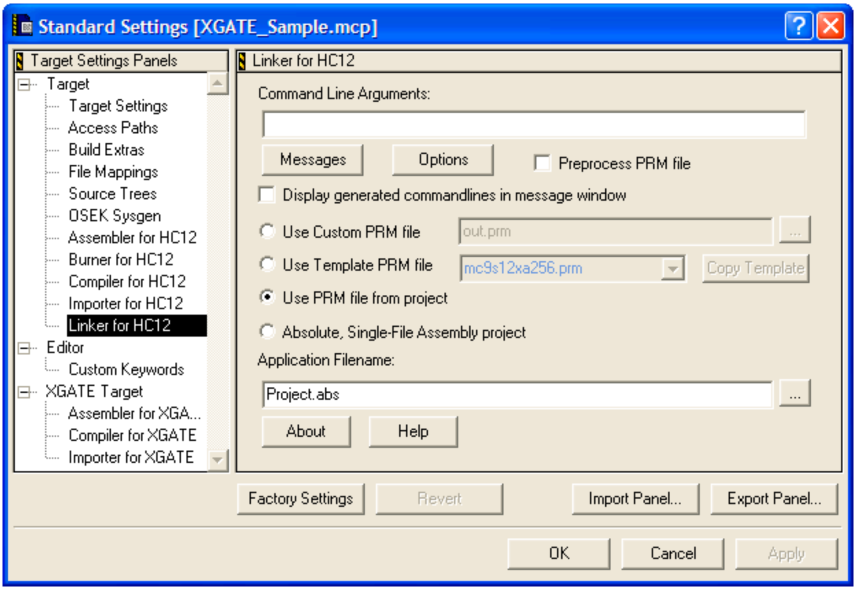
If you are using CodeWarrior to manage your project, a pre-configured PRM file for a particular derivative is already set up. The `.prm` file is located in the `prm` subfolder of your project folder.

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

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

The **Linker for HC12** preference panel allows you to select the PRM file you want to use for your CodeWarrior project. The default PRM file for a CodeWarrior project is the PRM file in the project window. To set preferences, select **Edit >** <*target_name*> **Settings > Target > Linker for HC12.** The **Linker for HC12** preference panel appears (Figure 1.28).

Figure 1.28 Linker for HC12 Preference Panel



Standard Settings [XGATE_Sample.mcp]

Linker for HC12

Command Line Arguments:

☐ Preprocess PRM file

☐ Display generated commandlines in message window

○ Use Custom PRM file: out.prm

○ Use Template PRM file: mc9s12xa256.prm

● Use PRM file from project

○ Absolute, Single-File Assembly project

Application Filename: Project.abs

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

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

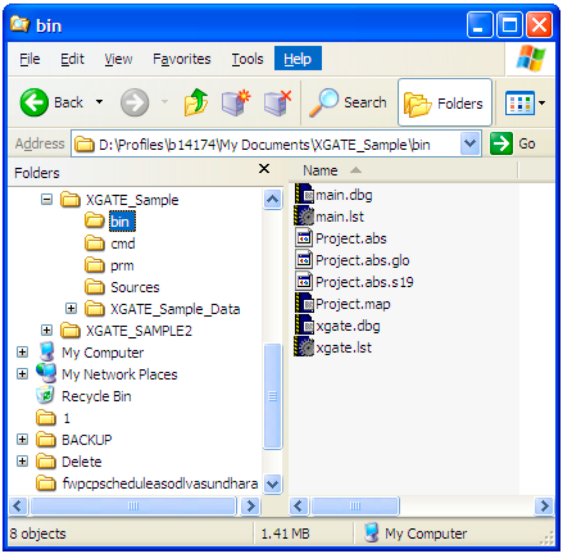
In case you want to change the filename of the application, you can determine the filename and its path with the *Application Filename:* text box. See the *Smart Linker section of the "Build Tools"* manual for details.

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

Linking Object-Code Files

You can run this relocatable assembly project from the **Project** menu: Select **Project > Make or Project > Debug.** The Linker generates a *.abs file and a *.abs.s19 standard S-Record File in the bin subfolder of the project directory. You can use the S-Record File for programming a ROM memory (Figure 1.29).

Figure 1.29 Project Directory in Windows Explorer After Linking



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

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

Linking with Linker

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

- Start your editor and create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as `<target_name>.prm`.
- Store the PRM file in a convenient location. A good spot would be directly into the project directory.
- In the `<target_name>.prm` file, add the name of the executable (`*.abs`) file, say `<target_name>.abs`. In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's `Project.prm` file **—** a PRM file for an MC9S12XA256 from another CodeWarrior project was adapted. Double-click on a .prm file to display contents.

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

NOTE The default size for the stack using the CodeWarrior New Project Wizard for the MC9S12XA256 is 256 bytes: (STACKSIZE 0x100). This command and `__SEG_END_SSTACK` in the assembly code file determine the size and placement of the stack in RAM:

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

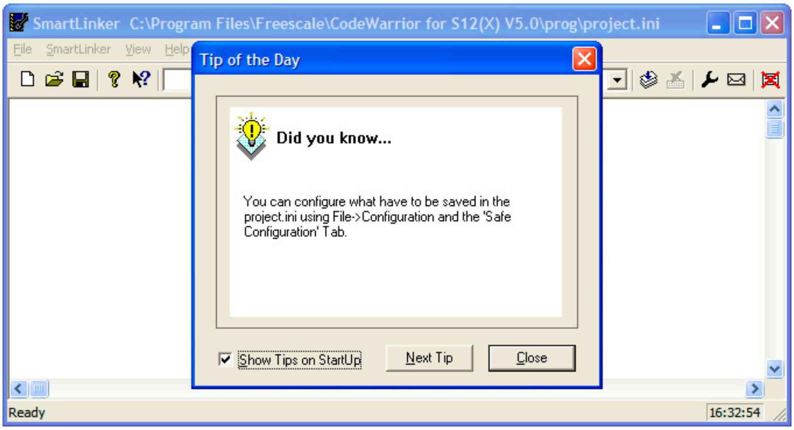
The commands in the linker parameter file are described in the Linker portion of the Build Tools manual.

To link a file, perform these steps:

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

   The **SmartLinker Default Configuration** windows appears (Figure 1.30).

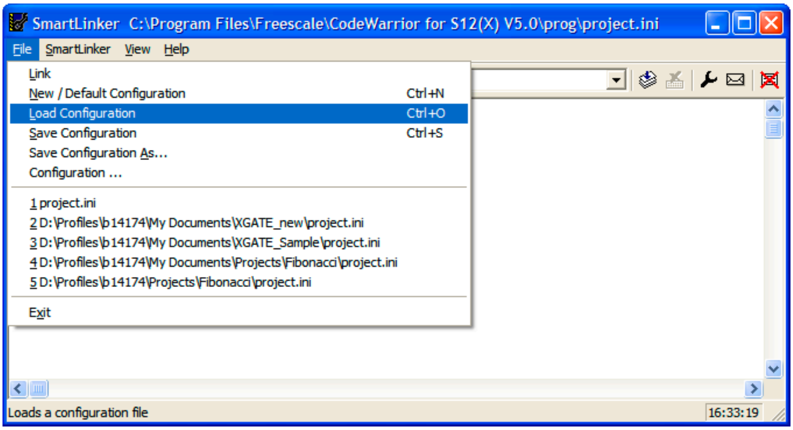
Figure 1.30 SmartLinker Default Configuration



You can configure what have to be saved in the project.ini using File->Configuration and the 'Safe Configuration' Tab.

2. Click the **Close** button to close the **Tip of the Day** dialog box.
3. Select **File > Load Configuration** (Figure 1.31) to load an existing project's configuration file.

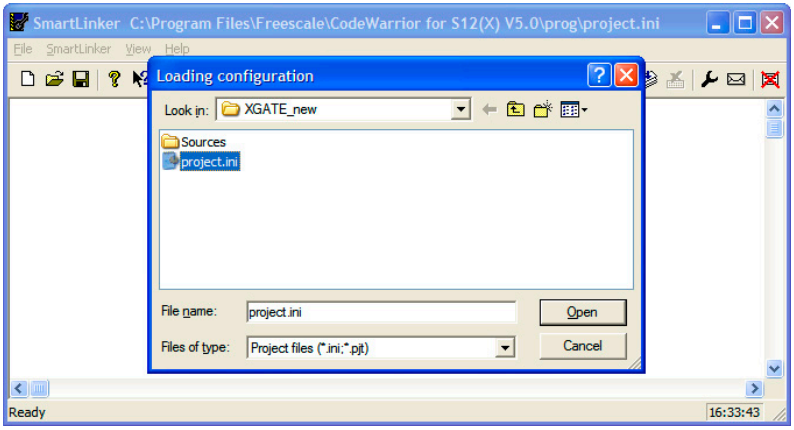
Figure 1.31 Load Configuration


SmartLinker C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\prog\project.ini
File
SmartLinker
View
Help
Link
New / Default Configuration Ctrl+N
Load Configuration Ctrl+O
Save Configuration Ctrl+S
Save Configuration As...
Configuration ...
1 project.ini
2 D:\Profiles\b14174\My Documents\XGATE_new\project.ini
3 D:\Profiles\b14174\My Documents\XGATE_Sample\project.ini
4 D:\Profiles\b14174\My Documents\Projects\Fibonacci\project.ini
5 D:\Profiles\b14174\Projects\Fibonacci\project.ini
Exit
16:33:19

Loads a configuration file

4. In the **Loading configuration** dialog box, select the same `<project>.ini` that the Assembler used for its configuration - the `project.ini` file in the project directory (Figure 1.32).

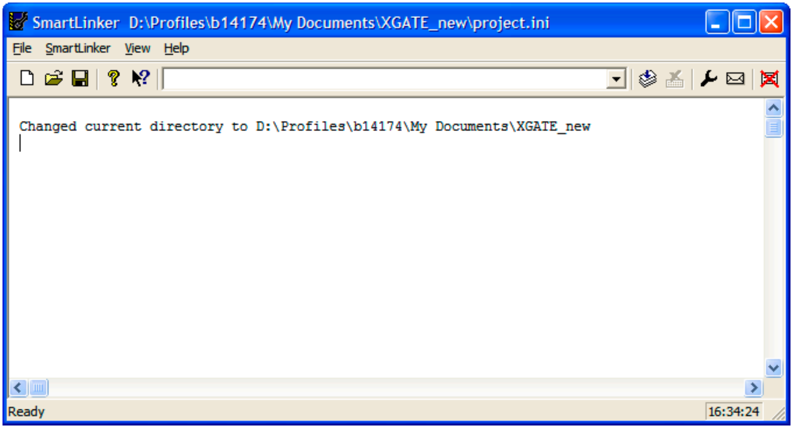
Figure 1.32 Loading configuration Dialog Box


SmartLinker C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\prog\project.ini
File SmartLinker View Help
Loading configuration
Look in: XGATE_new
Sources
project.ini
File name: project.ini
Files of type: Project files (*.ini;*.pjt)
Open
Cancel
Ready
16:33:43

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

   The current directory is changed (Figure 1.33).

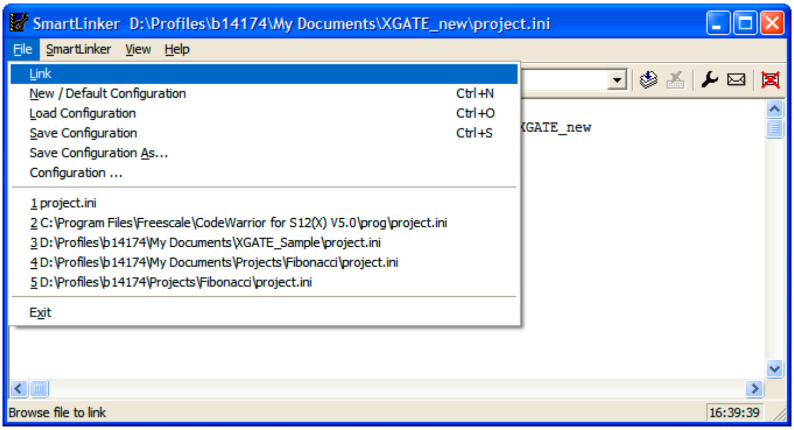
Figure 1.33 SmartLinker Configuration


SmartLinker D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File SmartLinker View Help
Ready
16:34:24

```
Changed current directory to D:\Profiles\b14174\My Documents\XGATE_new
```

6. Select **File > Link** (Figure 1.34).

Figure 1.34 Select File to Link

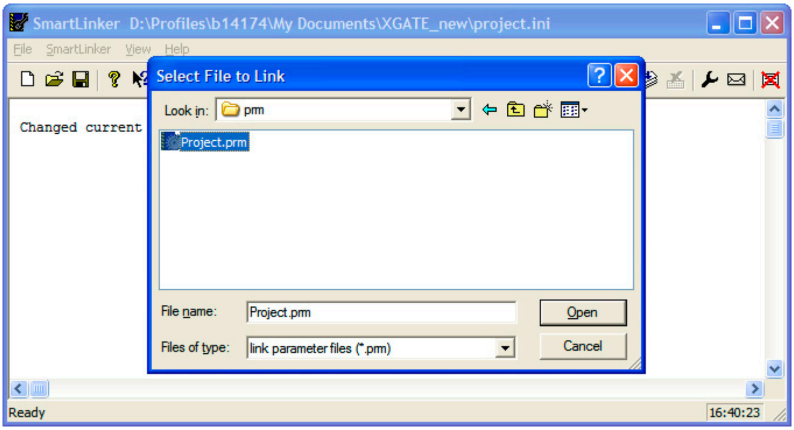

SmartLinker D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
SmartLinker
View
Help
Link
New / Default Configuration Ctrl+N
Load Configuration Ctrl+O
Save Configuration Ctrl+S
Save Configuration As...
Configuration ...
1 project.ini
2 C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\prog\project.ini
3 D:\Profiles\b14174\My Documents\XGATE_Sample\project.ini
4 D:\Profiles\b14174\My Documents\Projects\Fibonacci\project.ini
5 D:\Profiles\b14174\Projects\Fibonacci\project.ini
Exit
Browse file to link
16:39:39

```
XGATE_new
```

The **Select Files to Link** dialog box appears (Figure 1.35).

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

Figure 1.35 Select Files to Link Dialog Box


SmartLinker D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
SmartLinker
View
Help
Select File to Link
Open
Cancel
Ready
16:40:23

```
Changed current
```

Look in: prm

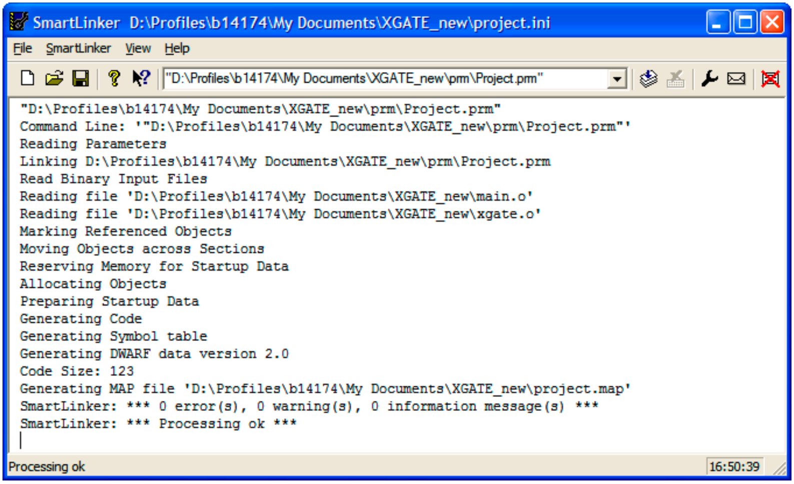
Project.prm

File name: Project.prm

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

8. Press the **Open** button.
   The SmartLinker links the object-code files in the `NAMES` section to produce the executable `*.abs` file as specified in the `LINK` portion of the Linker PRM file (Figure 1.36).

Figure 1.36 Linker Main Window After Linking


SmartLinker D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File SmartLinker View Help
"D:\Profiles\b14174\My Documents\XGATE_new\prm\Project.prm"
Processing ok
16:50:39

```
"D:\Profiles\b14174\My Documents\XGATE_new\prm\Project.prm"
Command Line: '"D:\Profiles\b14174\My Documents\XGATE_new\prm\Project.prm"'
Reading Parameters
Linking D:\Profiles\b14174\My Documents\XGATE_new\prm\Project.prm
Read Binary Input Files
Reading file 'D:\Profiles\b14174\My Documents\XGATE_new\main.o'
Reading file 'D:\Profiles\b14174\My Documents\XGATE_new\xgate.o'
Marking Referenced Objects
Moving Objects across Sections
Reserving Memory for Startup Data
Allocating Objects
Preparing Startup Data
Generating Code
Generating Symbol table
Generating DWARF data version 2.0
Code Size: 123
Generating MAP file 'D:\Profiles\b14174\My Documents\XGATE_new\project.map'
SmartLinker: *** 0 error(s), 0 warning(s), 0 information message(s) ***
SmartLinker: *** Processing ok ***
```

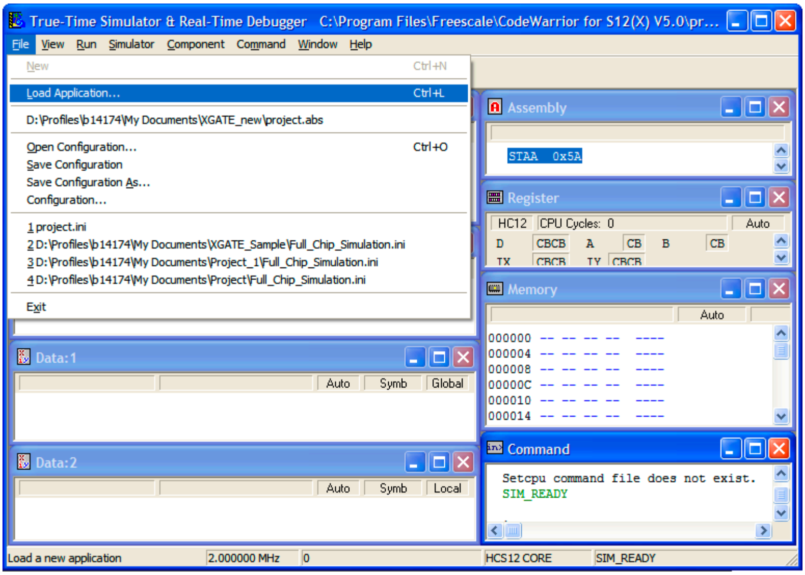
The messages in the linker’s project window indicate:

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

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

- Start the Simulator.
- Load the absolute executable file (Figure 1.37):
  - *File > Load Application...* and browse to the appropriate `*.abs` file, or
  - Select the given path to the executable file:

Figure 1.37 XGATE Simulator: Load Application


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\pr...
File View Run Simulator Component Command Window Help
New Ctrl+N
Load Application... Ctrl+L
D:\Profiles\b14174\My Documents\XGATE_new\project.abs
Open Configuration... Ctrl+O
Save Configuration
Save Configuration As...
Configuration...
1 project.ini
2 D:\Profiles\b14174\My Documents\XGATE_Sample\Full_Chip_Simulation.ini
3 D:\Profiles\b14174\My Documents\Project_1\Full_Chip_Simulation.ini
4 D:\Profiles\b14174\My Documents\Project\Full_Chip_Simulation.ini
Exit
Assembly
Register
Memory
Command
Data:1
Data:2
Auto
Symb
Global
Local

```
STAA  0x5A
```

```
HC12  CPU Cycles: 0                Auto
D   CBCB   A   CB   B   CB
TX  CBCB   TY  CBCB
```

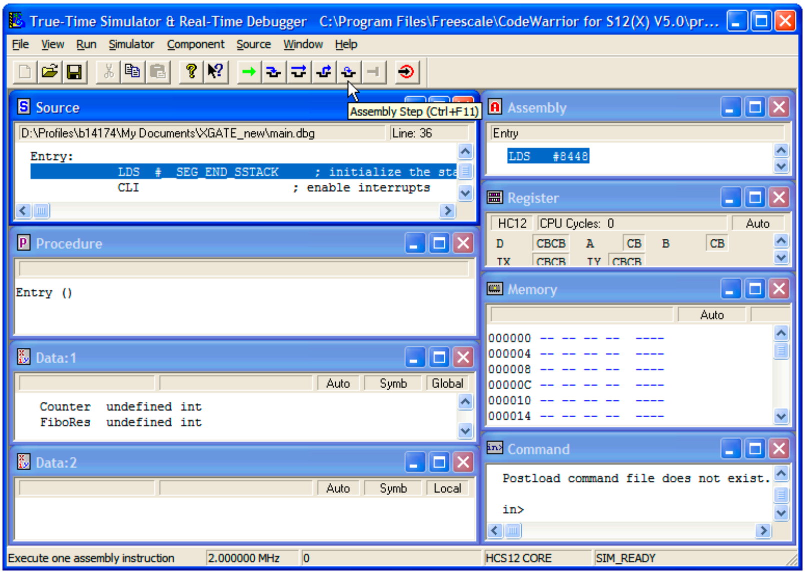
```
000000 -- -- -- --  ----
000004 -- -- -- --  ----
000008 -- -- -- --  ----
00000C -- -- -- --  ----
000010 -- -- -- --  ----
000014 -- -- -- --  ----
```

```
Setcpu command file does not exist.
SIM_READY
```

Load a new application | 2.000000 MHz | 0 | HCS12 CORE | SIM_READY

- Assembly-Step (Figure 1.38) through the program source code (or do something else...).

Figure 1.38 Assembly Stepping


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\pr...
File View Run Simulator Component Source Window Help
Assembly Step (Ctrl+F11)
Source
Procedure
Data:1
Data:2
Assembly
Register
Memory
Command
Auto
Symb
Global
Local

```
D:\Profiles\b14174\My Documents\XGATE_new\main.dbg    Line: 36
Entry:
            LDS  #  SEG_END_SSTACK      ; initialize the sta
            CLI                        ; enable interrupts
```

```
Entry ()
```

```
Counter  undefined int
FiboRes  undefined int
```

```
Entry
  LDS    #8448
```

```
HC12  CPU Cycles:  0                 Auto
D     CBCB   A   CB    B    CB
IX    CBCB   IY  CBCB
```

```
000000 -- -- -- --  ----
000004 -- -- -- --  ----
000008 -- -- -- --  ----
00000C -- -- -- --  ----
000010 -- -- -- --  ----
000014 -- -- -- --  ----
```

```
Postload command file does not exist.

in>
```

```
Execute one assembly instruction   2.000000 MHz   0   HCS12 CORE   SIM_READY
```

Directly Generating ABS File

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

NOTE The assembler for the Philips XA does not support the ELF format. Directly generating an ABS file is only possible in ELF.

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

Generating ABS File Using CodeWarrior

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

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

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

Single Absolute-Assembly main.asm File

Only one `*.asm` assembly source-code file can be used in an absolute-assembly project. The `main.asm` source code file differs slightly from a file used in relocatable assembly (Listing 1.4).

Listing 1.4 main.asm file - absolute assembly

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

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

; common defines and macros
            INCLUDE 'derivative.inc'

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

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


; code section
            ORG   ROMStart
```

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

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


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

Pay special attention to the following points:

- The Reset vector is usually initialized in the assembly source file with the application entry point. An absolute section containing the application’s entry point address is created at the Reset vector address. To set the entry point of the application at address `$FFFE` on the `Entry` symbol, the following code is used (Listing 1.6):

Listing 1.5 Using ORG to set the Reset vector

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

- The ABSENTRY directive is used to write the address of the application entry point in the generated absolute file. To set the entry point of the application on the Entry label in the absolute file, the following code is used (Listing 1.6).

Listing 1.6 Using ABSENTRY to enter the entry-point address

```
ABSENTRY Entry
```

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

Assembling main.asm

From the *Project* menu, select *Bring Up To Date* or select the main.asm file in the project window and select *Compile*. If the project's preferences are set to create an assembler output listing file, this will generate a listing file as shown in Listing 1.7.

Listing 1.7 Assembler output listing file of main.asm

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

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      ;************************************
    2    2                      ;* This stationery serves as the fram
    3    3                      ;* user application (single file, abs
    4    4                      ;* For a more comprehensive program t
    5    5                      ;* demonstrates the more advanced fun
    6    6                      ;* processor, please see the demonstr
    7    7                      ;* located in the examples subdirecto
    8    8                      ;* Freescale CodeWarrior for the HC1
    9    9                      ;************************************
   10   10
   11   11                      ; export symbols
   12   12                                  XDEF Entry            ; e
   13   13                                  ABSENTRY Entry        ; f
```

```
  14   14                                                        ; a
  15   15
  16   16                      ; include derivative specific macros
  17   17                                  INCLUDE 'mc9s12c32.inc'
5396   18
5397   19          0000 4000   ROMStart    EQU   $4000  ; absolute a
5398   20
5399   21                      ; variable/data section
5400   22                                  ORG RAMStart          ; I
5401   23  a000800             Counter     DS.W 1
5402   24  a000802             FiboRes     DS.W 1
5403   25
5404   26
5405   27                      ; code section
5406   28                                  ORG   ROMStart
5407   29                      Entry:
5408   30  a004000 CF10 00                 LDS   #RAMEnd+1      ; i
5409   31  a004003 10EF                    CLI                   ;
5410   32                      mainLoop:
5411   33  a004005 CE00 01                 LDX   #1              ; X
5412   34                      counterLoop:
5413   35  a004008 7E08 00                 STX   Counter         ; u
5414   36  a00400B 070E                    BSR   CalcFibo
5415   37  a00400D 7C08 02                 STD   FiboRes         ; s
5416   38  a004010 FE08 00                 LDX   Counter
5417   39  a004013 08                      INX
5418   40  a004014 8E00 18                 CPX   #24             ; L
5419   41  a004017 26EF                    BNE   counterLoop
5420   42  a004019 20EA                    BRA   mainLoop        ; r
5421   43
5422   44                      CalcFibo:  ; Function to calculate Fi
5423   45  a00401B CD00 00                 LDY   #$00            ; s
5424   46  a00401E CC00 01                 LDD   #$01            ; l
5425   47  a004021 0405 07                 DBEQ  X,FiboDone      ; l
5426   48                      FiboLoop:
5427   49  a004024 19EE                    LEAY  D,Y             ; o
5428   50  a004026 B7C6                    EXG   D,Y             ; e
5429   51  a004028 0435 F9                 DBNE  X,FiboLoop
5430   52                      FiboDone:
5431   53  a00402B 3D                      RTS                   ; r
5432   54
5433   55
5434   56                      ;************************************
5435   57                      ;*                 I
5436   58                      ;************************************
5437   59                                  ORG   $FFFE
```

```
 5438   60  a00FFFE 4000                     DC.W  Entry           ; R
```

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

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

Using Assembler for Absolute Assembly

Create a new configuration *project.ini* file and directory for the absolute assembly project using the standalone Assembler Build Tool. This section does not go into the detail that was done for the relocatable assembly section. Use an absolute assembly source file of the type listed in Listing 1.8.

Listing 1.8 Main.asm file for absolute assembly

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

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

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

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

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

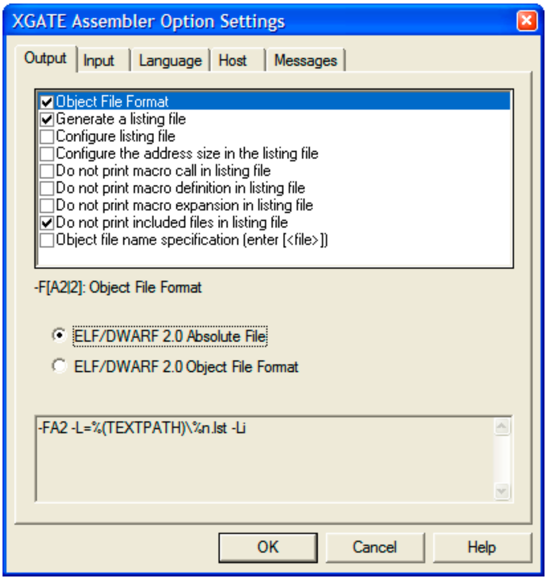
```
; code section
            ORG   ROMStart
Entry:
            LDS   #RAMEnd+1       ; initialize the stack pointer to
                                  ; highest absolute RAM address
            CLI                   ; enable interrupts
mainLoop:
            LDX   #1              ; X contains counter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

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


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

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

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

- Select *Assembler* > *Options*. The *XGATE Assembler Option Settings* dialog box appears (Figure 1.39).

Figure 1.39 XGATE Assembler Option Settings dialog box



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

-F[A2|2]: Object File Format

● ELF/DWARF 2.0 Absolute File
○ ELF/DWARF 2.0 Object File Format

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

- In the *Output* panel, select the check box in front of *Object File Format*. The Assembler displays more information at the bottom of the dialog box. Select the *ELF/DWARF 2.0 Absolute File* radio button. The assembler options for generating a listing file can also be set at this point, if desired. Click *OK*.
- Select the assembly source-code file that will be assembled: *Select File* > *Assemble*. The *Select File to Assemble* dialog box appears.
- Browse to the assembly source-code file. Click *Open*. The Assembler now assembles the source code. Error-message or positive feedback about the assembly process is created in the assembler main window.

Make sure that the `GENPATH` configuration is set for the include file used by the `main.asm` file in this project in the event an error message for a missing file appears.

Confer “Adding GENPATH” on page 45 for instructions for setting a GENPATH. (*File* > *Configuration...* > *Environment* > *General Path* and browse for the missing include file.) After adding a GENPATH to the folder of the include file, try assembling again.

Select *File* > *Assemble* and browse for the `*.asm` file and press *Open* for the assembly command. This time, it should assemble correctly.

The messages indicate that:

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

The `main.abs` file can also be used as input to the Simulator/Debugger - another Build Tool in the CodeWarrior Development Studio for S12(X) v5.x, with which you can follow the execution of your program.

2

Assembler Graphical User Interface

The Macro Assembler runs under Microsoft® Windows® 2000, Windows® XP, or Windows Vista™ operating systems.

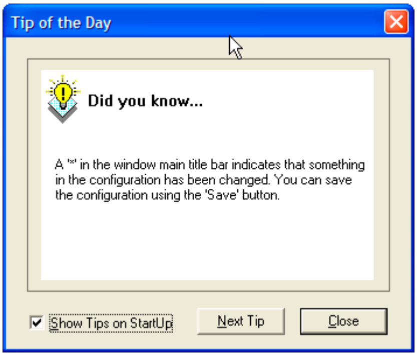
This chapter covers the following topics:

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

Starting Assembler

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

Figure 2.1 Tip of the Day Dialog Box



Did you know...

A '*' in the window main title bar indicates that something in the configuration has been changed. You can save the configuration using the 'Save' button.

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

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

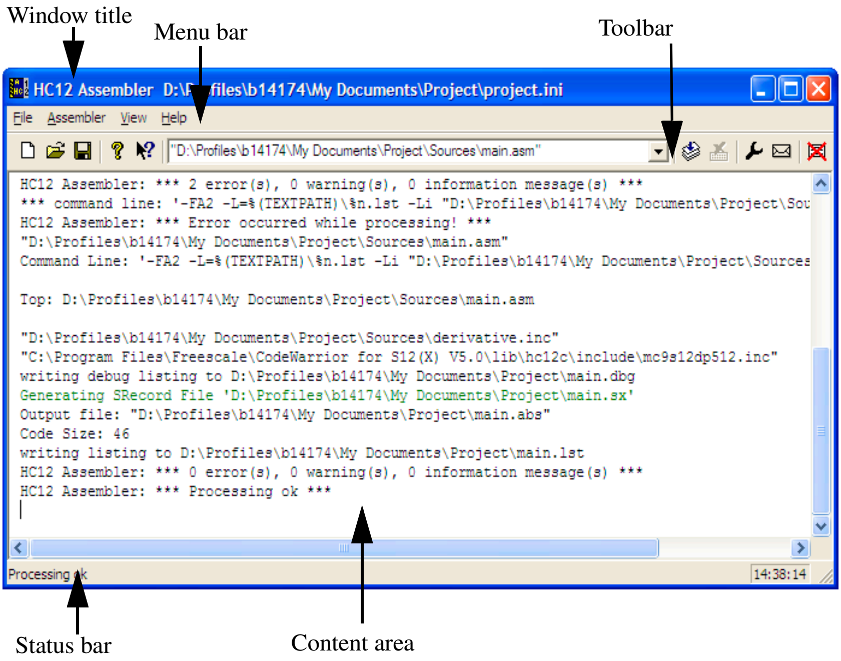
NOTE If you want the Assembler to automatically open the standard **Tip of the Day** dialog box at Assembler start up, choose **Help > Tip of the Day**. The Assembler displays the **Tip of the Day** dialog box. Check the Show Tips on StartUp checkbox.

Assembler Main Window

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

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

Figure 2.2 Assembler Main Window


Window title
Menu bar
Toolbar
HC12 Assembler D:\Profiles\b14174\My Documents\Project\project.ini
File Assembler View Help
"D:\Profiles\b14174\My Documents\Project\Sources\main.asm"
Processing ok
14:38:14
Status bar
Content area

```
HC12 Assembler: *** 2 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-FA2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\Project\Sou
HC12 Assembler: *** Error occurred while processing! ***
"D:\Profiles\b14174\My Documents\Project\Sources\main.asm"
Command Line: '-FA2 -L=%(TEXTPATH)\%n.lst -Li "D:\Profiles\b14174\My Documents\Project\Sources

Top: D:\Profiles\b14174\My Documents\Project\Sources\main.asm

"D:\Profiles\b14174\My Documents\Project\Sources\derivative.inc"
"C:\Program Files\Freescale\CodeWarrior for S12(X) V5.0\lib\hc12c\include\mc9s12dp512.inc"
writing debug listing to D:\Profiles\b14174\My Documents\Project\main.dbg
Generating SRecord File 'D:\Profiles\b14174\My Documents\Project\main.sx'
Output file: "D:\Profiles\b14174\My Documents\Project\main.abs"
Code Size: 46
writing listing to D:\Profiles\b14174\My Documents\Project\main.lst
HC12 Assembler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
HC12 Assembler: *** Processing ok ***
```

Window Title

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

Content Area

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

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

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

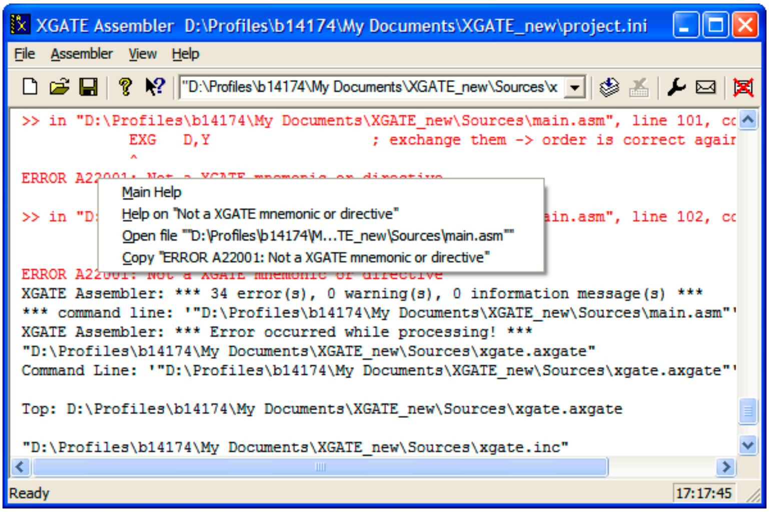
All text in the assembler window content area can have context information consisting of two items:

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

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

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

Figure 2.3 Right-Context Help


XGATE Assembler D:\Profiles\b14174\My Documents\XGATE_new\project.ini
File
Assembler
View
Help
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\x
Main Help
Help on "Not a XGATE mnemonic or directive"
Open file ""D:\Profiles\b14174\M...TE_new\Sources\main.asm""
Copy "ERROR A22001: Not a XGATE mnemonic or directive"
Ready
17:17:45

```
>> in "D:\Profiles\b14174\My Documents\XGATE_new\Sources\main.asm", line 101, cc
            EXG   D,Y                 ; exchange them -> order is correct agair
            ^
ERROR A22001: Not a XGATE mnemonic or directive

>> in "D:                                                       ain.asm", line 102, cc


ERROR A22001: Not a XGATE mnemonic or directive
XGATE Assembler: *** 34 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '"D:\Profiles\b14174\My Documents\XGATE_new\Sources\main.asm"'
XGATE Assembler: *** Error occurred while processing! ***
"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"
Command Line: '"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate"'

Top: D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.axgate

"D:\Profiles\b14174\My Documents\XGATE_new\Sources\xgate.inc"
```

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

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

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

Toolbar

Figure 2.4 displays the elements of the Toolbar.

Figure 2.4 Toolbar

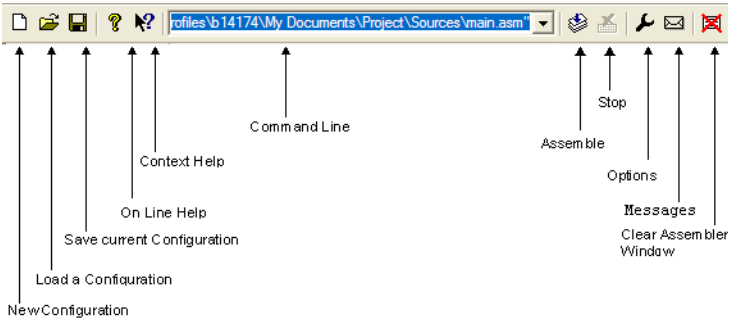


Table 2.1 describes the Assembler toolbar elements.

Table 2.1 Assemble Toolbar Elements

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

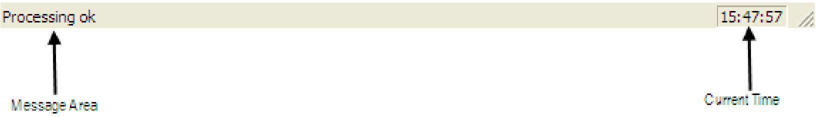
Table 2.1 Assemble Toolbar Elements

Element	Decription
↖?	Context Help — Click to invoke the context help.
"D:\Profiles\b14174\My ▾	Editable combo box — Click to display the list of commands which were executed.
	Assemble — Click to execute command line.
	Stop — Click to stop the current assembly session; is enabled when some file is assembled.
🔧	Options — Click to open the **Option Settings** dialog box.
✉	Message — Click to open the **Message Settings** dialog box.
☒	Clear — Click to clear the assembler window's content area.

Status Bar

Figure 2.5 displays the elements of the Status bar.

Figure 2.5 Status Bar



Processing ok

15:47:57

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

Assembler Menu Bar

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

Table 2.2 Menu Bar Options

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

File Menu

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

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

Table 2.3 File Menu Options

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

Assembler Menu

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

Table 2.4 Assembler Menu Options

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

View Menu

The View menu (Table 2.5) lets you customize the assembler window. You can specify if the status bar or the toolbar must be displayed or be hidden. You can also define the font used in the window or clear the window.

Table 2.5 View Menu Options

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

Editor Settings Dialog Box

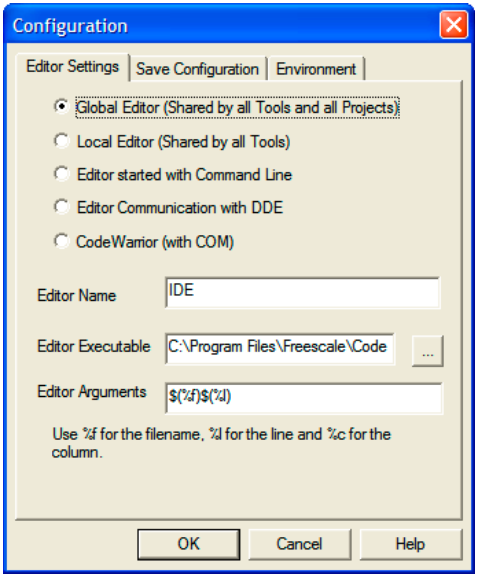
The Editor Setting dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

These are the following main entries:

Global Editor (Shared by all Tools and Projects)

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

Figure 2.6 Global Editor Configuration



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

Editor Name: IDE
Editor Executable: C:\Program Files\Freescale\Code
Editor Arguments: $(%f)$(%l)

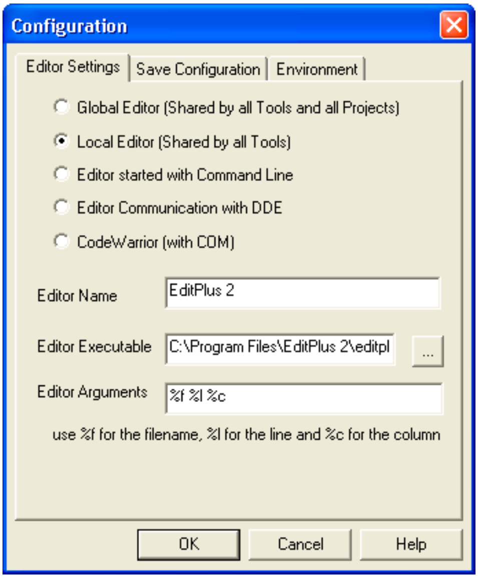
Use %f for the filename, %l for the line and %c for the column.

Local Editor (Shared by all Tools)

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

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

Figure 2.7 Local Editor Configuration



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

Editor Name: EditPlus 2
Editor Executable: C:\Program Files\EditPlus 2\editpl
Editor Arguments: %f %l %c

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

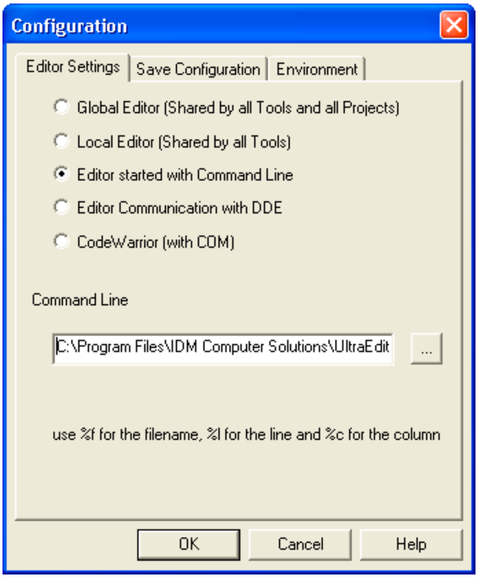
Editor Started with the Command Line

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

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

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

Figure 2.8 Command-Line Editor Configuration



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

Command Line

C:\Program Files\IDM Computer Solutions\UltraEdit

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

Examples of Configuring a Command-Line Editor

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

Listing 2.1 CodeWright Editor Configuration

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

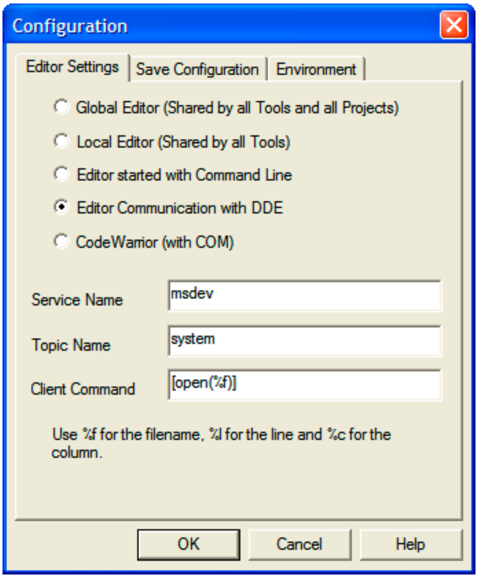
Listing 2.2 WinEdit Editor Configuration

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

Editor Started with DDE

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

Figure 2.9 DDE Editor Configuration



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

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

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

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

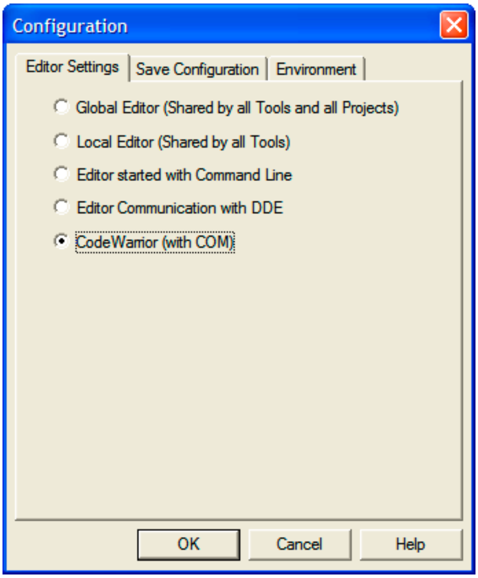
Listing 2.3 Microsoft Developer Studio Configuration Settings

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

CodeWarrior with COM

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

Figure 2.10 COM Editor Configuration



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

Modifiers

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

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

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

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

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

Save Configuration Dialog Box

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

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

This dialog box has the following configurations:

- *Options*: This item is related to the option and message settings. If this check box is set, the current option and message settings are stored in the project file when the configuration is saved. By disabling this check box, changes done to the option and message settings are not saved, and the previous settings remain valid.
- *Editor Configuration*: This item is related to the editor settings. If you set this check box, the current editor settings are stored in the project file when the configuration is saved. If you disable this check box, the previous settings remain valid.
- *Appearance*: This item is related to many parts like the window position (only loaded at startup time) and the command-line content and history. If you set this check box, these settings are stored in the project file when the current configuration is saved. If you disable this check box, the previous settings remain valid.
- *Environment Variables*: With this set, the environment variable changes done in the Environment property panel are also saved.

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

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

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

NOTE The configurations of the Assembler can, and in fact are intended to, coexist in the same file as the project configuration of other tools and the IDF. When an editor is configured by the shell, the assembler can read this content out of the project file, if present. The default project configuration filename is `project.ini`. The assembler automatically opens an existing `project.ini` in the current directory at startup. Also when using the -Prod: Specify project file at startup assembler option at startup or loading the configuration manually, a different name other than `project.ini` can be chosen.

Environment Configuration Dialog Box

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

The following variables are available:

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

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

The following buttons are available:

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

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

Option Settings Dialog Box

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

Table 2.6 Option Settings Options

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

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

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

The available options are listed in the Assembler Options chapter.

Message Settings Dialog Box

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

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

Table 2.7 Message Settings Options

Button	Description
Move to: Disabled	The selected messages are disabled; they will no longer be displayed.
Move to: Information	The selected messages are changed to information messages.

Table 2.7 Message Settings Options (*continued*)

Button	Description
Move to: Warning	The selected messages are changed to warning messages.
Move to: Error	The selected messages are changed to error messages.
Move to: Default	The selected messages are changed to their default message types.
Reset All	Resets all messages to their default message types.
OK	Exits this dialog box and saves any changes.
Cancel	Exits this dialog box without accepting any changes.
Help	Displays online help about this dialog box.

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

Table 2.8 Types of Message Groups

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

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

Changing the Class Associated with a Message

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

Example:

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

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

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

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

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

About... Dialog Box

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

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

Click on *OK* to close this dialog box.

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

Specifying Input File

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

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

Use the Toolbar Command Line to Assemble

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

Assembling a New File

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

Assembling a Previously Assembled File

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

Use the File > Assemble... Entry

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

Use Drag and Drop

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

Message/Error Feedback

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

Listing 2.4 Default Configuration of an Error Message

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

Listing 2.5 shows a typical error message.

Listing 2.5 Error Message Example

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

For different message formats, see the following Assembler options:

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

Use Information from the Assembler Window

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

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

Use a User-Defined Editor

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

Starting Editors by Specifying the Line Number in the Command line

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

Editors which Cannot by Started by Specifying the Line number on the Command Line

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

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

3

Environment

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

There are three ways to specify an environment:

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

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

Listing 3.1 Syntax for Setting Environment Variables

```
Parameter: KeyName=ParamDef
```

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

Listing 3.2 Setting the GENPATH Environment Variable

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

These parameters may be defined in several ways:

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

- Putting the definitions in a file given by the value of the ENVIRONMENT system environment variable.

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

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

NOTE The environment may also be changed using the -Env: Set environment variable assembler option.

Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool starts to search for files (e.g., for the default.env or .hidefaults)

Normally, the current directory of a launched tool is determined by the operating system or by the program that launches another one (e.g., IDE, Make Utility).

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

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

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

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

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

Environment Macros

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

Listing 3.3 Using a Macro for Setting Environment Variables

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

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

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

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

- {Compiler}

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

- {Project}

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

- {System}

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

Global Initialization File - mctools.ini (PC Only)

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

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

Listing 3.4 Usual Locations for the mcutools.ini Files

```
C:\WINDOWS\mcutools.ini
D:\Install Location\prog\mcutools.ini
```

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

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

Local Configuration File (Usually project.ini)

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

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

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

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

At startup, the configuration stored in the `project.ini` file located in the current directory is loaded. Local configuration file entries document the sections and entries you can put in a `project.ini` file.

Paths

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

Listing 3.5 Syntax used for Setting Path Lists of Environment Variables

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

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

Listing 3.6 Setting the Paths for the GEBNPATH Environment Variable

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

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

Listing 3.7 Recursive Search for a Continuation Line

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

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

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

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

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

Line Continuation

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

Listing 3.8 Using Multiple Lines for an Environment Variable

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

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

Listing 3.9 Alternate Form of using Multiple Lines

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

But this feature may cause problems when used together with paths (Listing 3.10).

Listing 3.10 Include Path with the Line Continuation Character

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

To avoid such problems, we recommend that you use a semicolon';' at the end of a path if there is a backslash '\' at the end (Listing 3.11).

Listing 3.11 Recommended Style when a Backlash is Present

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

Environment Variables Details

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

Table 3.1 Topics used for Describing Environment Variables

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

ABSPATH: Absolute file path

Description

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

Syntax

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

Tools

Compiler, Assembler, Linker, Decoder, or Debugger

Arguments

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

Example

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

ASMOPTIONS: Default assembler options

Description

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

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

Syntax

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

Tools

Assembler

Arguments

`<option>`: Assembler command-line option

Example

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

See also

Assembler Options chapter

COPYRIGHT: Copyright entry in object file

Description

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

Syntax

```
COPYRIGHT=<copyright>
```

Tools

Compiler, Assembler, Linker, or Librarian

Arguments

`<copyright>`: copyright entry

Example

```
COPYRIGHT=Copyright
```

See also

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

DEFAULTDIR: Default current directory

Description

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

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

Syntax

```
DEFAULTDIR=<directory>
```

Tools

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

Arguments

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

Example

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

See also

Current Directory

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

ENVIRONMENT: Environment file specification

Description

This variable has to be specified on the system level. Normally the Assembler looks in the current directory for an environment file named `default.env` (`.hidefaults` on UNIX). Using `ENVIRONMENT` (e.g., set in the `autoexec.bat` (DOS) or `.cshrc` (UNIX)), a different filename may be specified.

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

Syntax

```
ENVIRONMENT=<file>
```

Tools

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

Synonym

`HIENVIRONMENT`

Arguments

`<file>`: filename with path specification, without spaces

Example

```
ENVIRONMENT=\Freescale\prog\global.env
```

ERRORFILE: Filename specification error

Description

The `ERRORFILE` environment variable specifies the name for the error file (used by the Compiler or Assembler).

Possible format specifiers are:

- '`%n`': Substitute with the filename, without the path.
- '`%p`': Substitute with the path of the source file.
- '`%f`': Substitute with the full filename, i.e., with the path and name (the same as '`%p%n`').

In case of an improper error filename, a notification box is shown.

Syntax

```
ERRORFILE=<filename>
```

Tools

Compiler, Assembler, or Linker

Arguments

`<filename>`: Filename with possible format specifiers

Default

`EDOUT`

Examples

Listing 3.12 indicates error file in the current directory used to list all errors.

Listing 3.12 Naming an Error File

```
ERRORFILE=MyErrors.err
```

Listing 3.13 indicates error file in the `\tmp` directory used to list all errors.

Listing 3.13 Naming an Error File in a Specific Directory

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

Listing 3.14 indicates how to list all errors into a file with the same name as the source file, but with extension `*.err` and place in the same directory as the source file. For example, if we compile a file named `\sources\test.c`, an error list file `\sources\test.err` will be generated.

Listing 3.14 Naming an Error File as Source Filename

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

Listing 3.15 indicates how to list errors into a file with the same name as the source file, but with extension `*.err` and place into specific directory. For a `test.c` source file, a `\dir1\test.err` error list file will be generated.

Listing 3.15 Naming an Error File as Source Filename in a Specific Directory

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

For a `\dir1\dir2\test.c` source file, a `\dir1\dir2\errors.txt` error list file will be generated (Listing 3.16).

Listing 3.16 Naming an Error File with Full Path

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

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

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

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

Another example (Listing 3.17) shows the usage of this variable to support correct error feedback with the WinEdit Editor which looks for an error file called `EDOUT`:

Listing 3.17 Configuring Error Feedback with WinEdit

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

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

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

NOTE Always set this variable if using the WinEdit Editor, otherwise the editor cannot find the `EDOUT` file.

GENPATH: Search path for input file

Description

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

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

Syntax

`GENPATH={<path>}`

Tools

Compiler, Assembler, Linker, Decoder, or Debugger

Synonym

`HIPATH`

Arguments

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

Example

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

INCLUDETIME: Creation time in the object file

Description

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

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

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

Syntax

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

Tools

Compiler, Assembler, Linker, or Librarian

Arguments

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

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

Default

```
ON
```

Example

```
INCLUDETIME=OFF
```

See also

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

OBJPATH: Object file path

Description

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

Syntax

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

Tools

Compiler, Assembler, Linker, or Decoder

Arguments

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

Example

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

SRECORD: S-Record type

Description

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

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

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

Syntax

```
SRECORD=<RecordType>
```

Tools

Assembler, Linker, or Burner

Arguments

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

Example

```
SRECORD=S2
```

TEXTPATH: Text file path

Description

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

Syntax

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

Tools

Compiler, Assembler, Linker, or Decoder

Arguments

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

Example

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

TMP: Temporary directory

Description

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

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

Syntax

```
TMP=<directory>
```

Tools

Compiler, Assembler, Linker, Debugger, or Librarian

Arguments

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

Example

```
TMP=C:\TEMP
```

See also

Current Directory

USERNAME: User name in object file

Description

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

Syntax

```
USERNAME=<user>
```

Tools

Compiler, Assembler, Linker, or Librarian

Arguments

`<user>`: Name of user

Example

```
USERNAME=PowerUser
```

See also

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

4

Files

This chapter covers these topics:

- Input Files
- Output Files

Input Files

Input files to the Assembler:

- Source Files
- Object Files

Source Files

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

Include Files

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

Output Files

Output files from the Assembler:

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

Object Files

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

Absolute Files

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

S-Record Files

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

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

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

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

Listing Files

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

Debug Listing Files

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

Error Listing File

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

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

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

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

Interactive Mode (Assembler Window Open)

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

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

Batch Mode (Assembler Window Closed)

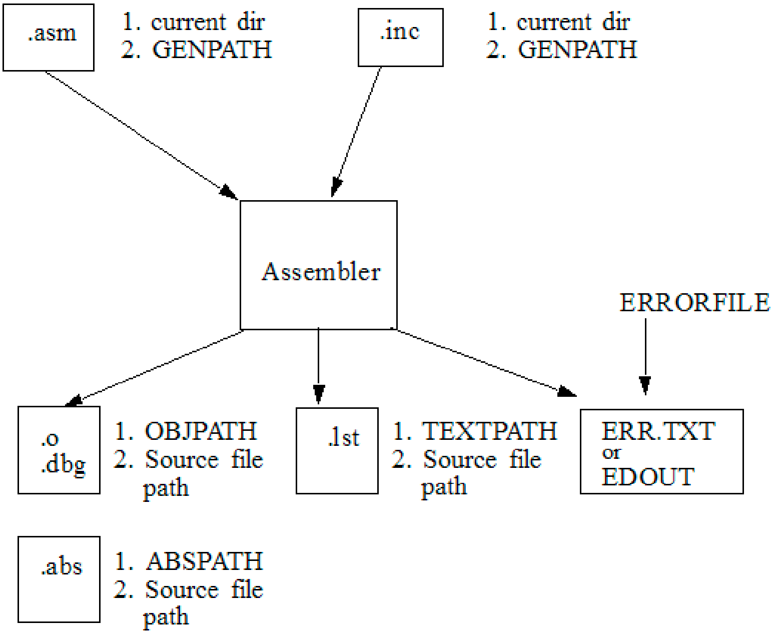
If ERRORFILE is set, the Assembler creates a message file named as specified in this environment variable.

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

File Processing

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

Figure 4.1 Files Used with the Assembler



5

Assembler Options

Types of Assembler Options

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

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

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

Table 5.1 ASMOPTIONS Environment Variable

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

Assembler options (Table 5.2) are grouped by:

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

Table 5.2 Assembler Option Categories

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

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

Each option has also a scope (Table 5.3)

Table 5.3 Scopes for Assembler Options

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

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

Assembler Option Details

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

Table 5.4 Assembler Option Details

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

Using Special Modifiers

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

The following modifiers are supported (Table 5.5)

Table 5.5 Special Modifiers for Assembler Options

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

Examples Using Special Modifiers

Listing 5.1 shows the assumed path and filename (filename base for the modifiers) used for the following examples.

Listing 5.1 Example Filename and Path for the Following Examples

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

Using the `%p` modifier displays the path with a file separator but without the filename:

`C:\Freescale\my demo\`

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

`TheWhole`

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

`TheWholeThing`

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

`myE`

To get the entire file extension, use the `%e` modifier:

`myExt`

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

`C:\Freescale\my demo\TheWholeThing`

The path in Listing 5.1 contains a space, therefore using `%"` or `%'` is recommended. Use `%"%f%"` or `%'%f%'` when there is a space in the path, filename, or extension:

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

or

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

Using `%(envVariable)`, an environment variable may be used. A file separator following `%(envVariable)` is ignored if the environment variable is empty or does not exist. If `TEXTPATH` is set as in Listing 5.2, then $(TEXTPATH)\myfile.txt is expressed as in Listing 5.3.

Listing 5.2 Example for Setting TEXTPATH

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

Listing 5.3 $(TEXTPATH)\myfile.txt Where TEXTPATH is Defined

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

However, if `TEXTPATH` does not exist or is empty, then $(TEXTPATH)\myfile.txt is expressed as in the following:

```
myfile.txt
```

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

```
myExt%
```

List of Assembler Options

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

Table 5.6 Assembler Options

Assembler Option
-Ci: Switch case sensitivity on label names OFF
-CMacAngBrack: Angle brackets for grouping macro arguments
-CMacBrackets: Square brackets for macro arguments grouping
-Compat: Compatibility modes
-D: Define label
-Env: Set environment variable
-F (-F2, -FA2) -F: Output file format
-H: Short help
-I: Include file path
-L: Generate a listing file
-Lasmc: Configure listing file
-Lasms: Configure the address size in the listing file
-Lc: No Macro call in listing file

Table 5.6 Assembler Options (*continued*)

Assembler Option
-Ld: No macro definition in listing file
-Le: No macro expansion in listing file
-Li: No included file in listing file
-Lic: License information
-LicA: License information about every feature in directory
-LicBorrow: Borrow license feature
-LicWait: Wait until floating license is available from floating license server
-MacroNest: Configure maximum macro nesting
-N: Display notify box
-NoBeep: No beep in case of an error
-NoDebugInfo: No debug information for ELF/DWARF files
-NoEnv: Do not use environment
-ObjN: Object filename specification
-Prod: Specify project file at startup
-Struct: Support for structured types
-V: Prints the assembler version
-View: Application standard occurrence
-W1: No information messages
-W2: No information and warning messages
-WErrFile: Create "err.log" error file
-Wmsg8x3: Cut filenames in Microsoft format to 8.3
-WmsgCE: RGB color for error messages
-WmsgCF: RGB color for fatal messages
-WmsgCI: RGB color for information messages
-WmsgCU: RGB color for user messages

Table 5.6 Assembler Options (*continued*)

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

Detailed Listing of All Assembler Options

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

-Ci: Switch case sensitivity on label names OFF

Description

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

Syntax

`-Ci`

Group

Input

Scope

Assembly Unit

Arguments

None

Default

None

Example

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

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

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

See also

-F (-F2, -FA2) -F: Output file format

-CMacAngBrack: Angle brackets for grouping macro arguments

Description

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

Syntax

```
-CMacAngBrack(ON|OFF)
```

Group

Language

Scope

Application

Arguments

`ON` or `OFF`

Default

None

See also

Macro Argument Grouping

-CMacBrackets: Square brackets for macro arguments grouping

-CMacBrackets: Square brackets for macro arguments grouping

Description

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

Syntax

```
-CMacBrackets(ON|OFF)
```

Group

Language

Scope

Application

Arguments

`ON` or `OFF`

Default

`ON`

See also

Macro Argument Grouping

-CMacAngBrack: Angle brackets for grouping macro arguments

-Compat: Compatibility modes

Description

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

- `=`: Operator `!=` means equal

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

- `!`: Support additional `!` operators

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

  `!^`: exponentiation

  `!m`: modulo

  `!@`: signed greater or equal

  `!g`: signed greater

  `!%`: signed less or equal

  `!t`: signed less than

  `!$`: unsigned greater or equal

  `!S`: unsigned greater

  `!&`: unsigned less or equal

  `!l`: unsigned less

  `!n`: one complement

  `!w`: low operator

  `!h`: high operator

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

- `c`: Alternate comment rules

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

- `s`: Symbol prefixes

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

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

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

- `$`: Supports the $ character in symbols

  With this suboption, the Assembler starts identifiers with a `$` sign.

- `a`: Add some additional directives

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

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

- `b`: support the `FOR` directive

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

Syntax

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

Group

Language

Scope

Application

Arguments

See description.

Default

None

Examples

Listing 5.4 demonstrates that when `-Compat=c`, comments can start with a `*`.

Listing 5.4 Comments Starting with an Asterisk (*)

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

When the `-Compat=c` assembler option is used, the first `DC.B` directive in Listing 5.5 has `"+ 1 , 1"` as a comment. A warning is issued because the "comment" does not start with a `";"` or a `"*"`. With `-Compat=c`, this code generates a warning and three bytes with constant values 1, 2, and 1. Without it, this code generates four 8-bit constants of 2, 1, 2, and 1.

Listing 5.5 Implicit Comment Start after Space

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

-D: Define label

Description

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

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

Syntax

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

Group

Input

Scope

Assembly Unit

Arguments

`<LabelName>`: Name of label.

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

Default

“`0`” for `Value`

Example

Conditional inclusion of a copyright notice. See Listing 5.6 and Listing 5.7.

Listing 5.6 Source Code that Conditionally Includes a Copyright Notice

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

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

When assembled with the options `-dADD_COPYRIGHT -dYEAR=2005`, Listing 5.7 is generated:

Listing 5.7 Generated List File

```
 1   1                     YearAsString:  MACRO
 2   2                         DC.B $30+(\1 /1000)%10
 3   3                         DC.B $30+(\1 / 100)%10
 4   4                         DC.B $30+(\1 /  10)%10
 5   5                         DC.B $30+(\1 /   1)%10
 6   6                     ENDM
 7   7
 8   8          0000 0001  ifdef ADD_COPYRIGHT
 9   9                       ORG $1000
```

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

-Env: Set environment variable

Description

This option sets an environment variable.

Syntax

`-Env<EnvironmentVariable>=<VariableSetting>`

Group

Host

Scope

Assembly Unit

Arguments

`<EnvironmentVariable>: Environment variable to be set`

`<VariableSetting>: Setting of the environment variable`

Default

None

Example

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

This is the same as:

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

in the `default.env` file.

See also

Environment Variables Details

-F (-F2, -FA2) -F: Output file format

Description

Define the format for the output file generated by the assembler.

With the option -F2 set, the assembler produces an ELF/DWARF object file. This object file format is also supported by other compiler, assembler and debugger vendors.

With the option -FA2 set, the assembler produces an ELF/DWARF absolute file. This object file format is also supported by other compiler, assembler and debugger vendors.

Syntax

-F (2 | A2)

Group

OUTPUT

Scope

Application

Arguments

2: ELF/DWARF 2.0 object file format (default)

A2: ELF/DWARF 2.0 absolute file format

Default

```
-F2
```

Example

```
ASMOPTIONS=-FA2
```

-H: Short help

Description

The -H option causes the Assembler to display a short list (i.e., help list) of available options within the assembler window. Options are grouped into Output, Input, Language, Host, Code Generation, Messages, and Various.

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

Syntax

-H

Group

Various

Scope

None

Arguments

None

Default

None

Example

Listing 5.8 is a portion of the list produced by the -H option:

Listing 5.8 Example Help Listing

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

-I: Include file path

Description

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

Syntax

```
-I<path>
```

Group

Input

Scope

None

Arguments

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

Default

None

Example

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

-L: Generate a listing file

Description

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

Syntax

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

Group

Output

Scope

Assembly unit

Arguments

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

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

Default

No generated listing file

Example

```
ASMOPTIONS=-L
```

In the following example assembly code, the macro `getadr16` can be used to load the 16-bit dress of variables. The `getadr16` macro takes two parameters: a register name and the variable name.

When the option -L is specified, the portion of code in Listing 5.9, together with the include file in Listing 5.10, generates the output in the assembly listing file shown in Listing 5.11.

Listing 5.9 Code with -L Option Specified

```
          XDEF entry
          INCLUDE "opt_l.inc"

myData:   SECTION
a:        DS.B  1

myConst:  SECTION
init:     DC.W  $1234

myCode:   SECTION
entry:
          getadr16 R2, init
          LDB      R4, (R2, R0)

          ADDL     R4, #1

          getadr16 R2, a
          STB      R4, (R2, R0)

loop:
          BRA loop
```

Listing 5.10 Include File

```
; getadr16 Dest Register, Variable Name
getadr16: MACRO
          LDL \1, #%XGATE_8(\2)
          LDH \1, #%XGATE_8_H(\2)
          ENDM
```

Listing 5.11 Output Generated in the Assembly Listing File

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                                XDEF entry
    2    2                                INCLUDE "opt_l.inc"
    3    1i                    ; getadr16 Dest Register, Variable Name
    4    2i                     getadr16: MACRO
    5    3i                               LDL \1, #%XGATE_8(\2)
    6    4i                               LDH \1, #%XGATE_8_H(\2)
```

```
   7    5i                                ENDM
   8    3
   9    4                       myData:   SECTION
  10    5   000000              a:        DS.B   1
  11    6
  12    7                       myConst:  SECTION
  13    8   000000 1234         init:     DC.W   $1234
  14    9
  15   10                       myCode:   SECTION
  16   11                       entry:
  17   12                                 getadr16 R2, init
  18    3m  000000 F2xx        +          LDL R2, #%XGATE_8(init)
  19    4m  000002 FAxx        +          LDH R2, #%XGATE_8_H(init)
  20   13   000004 6440                   LDB      R4, (R2, R0)
  21   14
  22   15   000006 E401                   ADDL     R4, #1
  23   16
  24   17                                 getadr16 R2, a
  25    3m  000008 F2xx        +          LDL R2, #%XGATE_8(a)
  26    4m  00000A FAxx        +          LDH R2, #%XGATE_8_H(a)
  27   18   00000C 7440                   STB      R4, (R2, R0)
  28   19
  29   20                       loop:
  30   21   00000E 3FFF                   BRA loop
```

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

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

See also

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

-Lasmc: Configure listing file

Description

The default-configured listing file shows a lot of information. With this option, the output can be reduced to columns which are of interest. This option configures which columns are printed in a listing file. To configure which lines to print, see the following assembler options:

- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: No included file in listing file

Syntax

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

Group

Output

Scope

Assembly unit

Arguments

`s` - Do not write the source column

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

`m` - Do not write the macro mark

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

`k` - Do not write the location type

`i` - Do not write the include mark column

`c` - Do not write the object code

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

Default

Write all columns.

Example

For the following assembly source code, the Assembler generates the default-configured output listing (Listing 5.12):

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

```
DC.B 0
```

Listing 5.12 Example Assembler Output Listing

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

In order to get this output without source file line numbers and other irrelevant parts for this `DC.B` example, the following option is added: `"-Lasmc=ramki"`. This generates the output listing in Listing 5.13:

Listing 5.13 Example Output Listing

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

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

See also

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

-Lasms: Configure the address size in the listing file

Description

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

- -Lc: No Macro call in listing file
- -Ld: No macro definition in listing file
- -Le: No macro expansion in listing file
- -Li: No included file in listing file

Syntax

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

Group

Output

Scope

Assembly unit

Arguments

1 - The address size is xx

2 - The address size is xxxx

3 - The address size is xxxxxx

4 - The address size is xxxxxxxx

Default

```
-Lasms3
```

Example

For the `NOP` instruction, the Assembler generates this default-configured output listing (Listing 5.14):

Listing 5.14 Example Assembler Output Listing

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

To change the size of the address column the `-Lasms1` option is added. This changes the address size to two digits.

Listing 5.15 Example Assembler Output Listing Configured with -Lasms1

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

See also

Assembler Listing File

Assembler options:

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

-Lc: No Macro call in listing file

Description

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

Syntax

`-Lc`

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-Lc`

In the following example assembly code, the macro `getadr16` can be used to load the 16-bit dress of variables. The `getadr16` macro takes two parameters: a register and the variable name.

When you specify the option `-Lc` using the portion of code shown in Listing 5.16, and the include file shown in Listing 5.17, the assembler generates the output in Listing 5.18 in the assembler listing file.

Listing 5.16 Code Using -Lc Option

```
          XDEF entry
          INCLUDE "opt_l.inc"

myData:   SECTION
a:        DS.B  1
```

```
myConst:  SECTION
init:     DC.W  $1234

myCode:   SECTION
entry:
          getadr16 R2, init
          LDB      R4, (R2, R0)

          ADDL     R4, #1

          getadr16 R2, a
          STB      R4, (R2, R0)

loop:
          BRA loop
```

Listing 5.17 Include File

```
; getadr16 Dest Register, Variable Name
getadr16: MACRO
          LDL \1, #%XGATE_8(\2)
          LDH \1, #%XGATE_8_H(\2)
          ENDM
```

Listing 5.18 Assembler Generated Output in the Assembly Listing File

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF entry
    2    2                               INCLUDE "opt_l.inc"
    3    1i                     ; getadr16 Dest Register, Variable Name
    4    2i                     getadr16: MACRO
    5    3i                              LDL \1, #%XGATE_8(\2)
    6    4i                              LDH \1, #%XGATE_8_H(\2)
    7    5i                              ENDM
    8    3
    9    4                      myData:   SECTION
   10    5   000000             a:        DS.B  1
   11    6
   12    7                      myConst:  SECTION
   13    8   000000 1234        init:     DC.W  $1234
   14    9
```

```
15   10                      myCode:   SECTION
16   11                      entry:
18    3m  000000 F2xx       +          LDL R2, #%XGATE_8(init)
19    4m  000002 FAxx       +          LDH R2, #%XGATE_8_H(init)
20   13   000004 6440                  LDB      R4, (R2, R0)
21   14
22   15   000006 E401                  ADDL     R4, #1
23   16
25    3m  000008 F2xx       +          LDL R2, #%XGATE_8(a)
26    4m  00000A FAxx       +          LDH R2, #%XGATE_8_H(a)
27   18   00000C 7440                  STB      R4, (R2, R0)
28   19
29   20                      loop:
30   21   00000E 3FFF                  BRA loop
```

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

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

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

See also

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

-Ld: No macro definition in listing file

Description

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

Syntax

`-Ld`

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-Ld`

In the following example assembly code, the macro `getadr16` can be used to load the 16-bit dress of variables. The `getadr16` macro takes two parameters: a register and the variable name.

When you specify the `-Ld` option, using the portion of code shown in Listing 5.19, together with the include file shown in Listing 5.20, the assembler generates the output shown in Listing 5.21 in the assembly listing file.

Listing 5.19 Code Using the -Ld Option

```
          XDEF entry
          INCLUDE "opt_l.inc"

myData:   SECTION
a:        DS.B   1
```

```
myConst:  SECTION
init:     DC.W  $1234

myCode:   SECTION
entry:
          getadr16 R2, init
          LDB      R4, (R2, R0)

          ADDL     R4, #1

          getadr16 R2, a
          STB      R4, (R2, R0)

loop:
          BRA loop
```

Listing 5.20 Include File

```
; getadr16 Dest Register, Variable Name
getadr16: MACRO
          LDL \1, #%XGATE_8(\2)
          LDH \1, #%XGATE_8_H(\2)
          ENDM
```

Listing 5.21 Assembler Generated Output in the Assembly Listing File

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF entry
    2    2                               INCLUDE "opt_l.inc"
    3   1i                     ; getadr16 Dest Register, Variable Name
    4   2i                      getadr16: MACRO
    8    3
    9    4                      myData:   SECTION
   10    5   000000             a:        DS.B  1
   11    6
   12    7                      myConst:  SECTION
   13    8   000000 1234        init:     DC.W  $1234
   14    9
   15   10                      myCode:   SECTION
   16   11                      entry:
   17   12                                getadr16 R2, init
```

```
18    3m   000000 F2xx        +          LDL R2, #%XGATE_8(init)
19    4m   000002 FAxx        +          LDH R2, #%XGATE_8_H(init)
20   13    000004 6440                   LDB      R4, (R2, R0)
21   14
22   15    000006 E401                   ADDL     R4, #1
23   16
24   17                                  getadr16 R2, a
25    3m   000008 F2xx        +          LDL R2, #%XGATE_8(a)
26    4m   00000A FAxx        +          LDH R2, #%XGATE_8_H(a)
27   18    00000C 7440                   STB      R4, (R2, R0)
28   19
29   20                        loop:
30   21    00000E 3FFF                   BRA loop
```

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

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

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

See also

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

-Le: No macro expansion in listing file

Description

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

Syntax

`-Le`

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-Le`

In the following example assembly code, the macro `getadr16` can be used to load the 16-bit dress of variables. The `getadr16` macro takes two parameters: a register and the variable name.

When you specify the `-Le` option, using the portion of code shown in Listing 5.22, together with the include file in Listing 5.23, the assembler generates the output shown in Listing 5.24 in the assembly listing file.

Listing 5.22 Code using -Le Option

```
          XDEF entry
          INCLUDE "opt_l.inc"

myData:   SECTION
a:        DS.B  1
```

```
myConst:  SECTION
init:     DC.W  $1234

myCode:   SECTION
entry:
          getadr16 R2, init
          LDB      R4, (R2, R0)

          ADDL     R4, #1

          getadr16 R2, a
          STB      R4, (R2, R0)

loop:
          BRA loop
```

Listing 5.23 Include File

```
; getadr16 Dest Register, Variable Name
getadr16: MACRO
          LDL \1, #%XGATE_8(\2)
          LDH \1, #%XGATE_8_H(\2)
          ENDM
```

Listing 5.24 Assembler Generated Output in Assembly Listing File

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                               XDEF entry
    2    2                               INCLUDE "opt_l.inc"
    3    1i                     ; getadr16 Dest Register, Variable Name
    4    2i                     getadr16: MACRO
    5    3i                              LDL \1, #%XGATE_8(\2)
    6    4i                              LDH \1, #%XGATE_8_H(\2)
    7    5i                              ENDM
    8    3
    9    4                      myData:   SECTION
   10    5   000000             a:        DS.B  1
   11    6
   12    7                      myConst:  SECTION
   13    8   000000 1234        init:     DC.W  $1234
   14    9
```

```
  15   10                          myCode:   SECTION
  16   11                          entry:
  17   12                                    getadr16 R2, init
  20   13   000004 6440                      LDB      R4, (R2, R0)
  21   14
  22   15   000006 E401                      ADDL     R4, #1
  23   16
  24   17                                    getadr16 R2, a
  27   18   00000C 7440                      STB      R4, (R2, R0)
  28   19
  29   20                          loop:
  30   21   00000E 3FFF                      BRA loop
```

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

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

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

See also

-L: Generate a listing file

-Lc: No Macro call in listing file

-Ld: No macro definition in listing file

-Li: No included file in listing file

-Li: No included file in listing file

Description

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

Syntax

```
-Li
```

Group

Output

Scope

Assembly unit

Arguments

None

Default

None

Example

`ASMOPTIONS=-Li`

In the following example of assembly code, the macro `getadr16` can be used to load the 16-bit dress of variables. The `getadr16` macro takes two parameters: a register and the variable name.

When you specify the `-Li` option, using the portion of code shown in Listing 5.25, together with the include file shown in Listing 5.26, the assembler generates the output shown in Listing 5.27 in the assembly listing file.

Listing 5.25 Code using -Li Option

```
          XDEF entry
          INCLUDE "opt_l.inc"

myData:   SECTION
a:        DS.B  1

myConst:  SECTION
init:     DC.W  $1234

myCode:   SECTION
entry:
          getadr16 R2, init
          LDB      R4, (R2, R0)

          ADDL     R4, #1

          getadr16 R2, a
          STB      R4, (R2, R0)

loop:
          BRA loop
```

Listing 5.26 Include File

```
; getadr16 Dest Register, Variable Name
getadr16: MACRO
          LDL \1, #%XGATE_8(\2)
          LDH \1, #%XGATE_8_H(\2)
          ENDM
```

Listing 5.27 Assembler Generated Output in the Assembly Listing File

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                                XDEF entry
    2    2                                INCLUDE "opt_l.inc"
    8    3
    9    4                      myData:   SECTION
   10    5   000000             a:        DS.B  1
   11    6
   12    7                      myConst:  SECTION
   13    8   000000 1234        init:     DC.W  $1234
   14    9
   15   10                      myCode:   SECTION
   16   11                      entry:
   17   12                                getadr16 R2, init
   18    3m  000000 F2xx       +          LDL R2, #%XGATE_8(init)
   19    4m  000002 FAxx       +          LDH R2, #%XGATE_8_H(init)
   20   13   000004 6440                  LDB      R4, (R2, R0)
   21   14
   22   15   000006 E401                  ADDL     R4, #1
   23   16
   24   17                                getadr16 R2, a
   25    3m  000008 F2xx       +          LDL R2, #%XGATE_8(a)
   26    4m  00000A FAxx       +          LDH R2, #%XGATE_8_H(a)
   27   18   00000C 7440                  STB      R4, (R2, R0)
   28   19
   29   20                      loop:
   30   21   00000E 3FFF                  BRA loop
```

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

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

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

See also

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

-Lic: License information

Description

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

Syntax

```
-Lic
```

Group

Various

Scope

None

Arguments

None

Default

None

Example

```
ASMOPTIONS=-Lic
```

See also

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

-LicA: License information about every feature in directory

Description

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

Syntax

```
-LicA
```

Group

Various

Scope

None

Arguments

None

Default

None

Example

```
ASMOPTIONS=-LicA
```

See also

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

-LicBorrow: Borrow license feature

Description

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

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

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

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

Syntax

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

Group

Host

Scope

None

Arguments

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

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

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

Default

None

Defines

None

Pragmas

None

Example

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

See also

Assembler options:

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

-LicWait: Wait until floating license is available from floating license server

Description

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

Syntax

```
-LicWait
```

Group

Host

Scope

None

Arguments

None

Default

None

Example

```
ASMOPTIONS=-LicWait
```

See also

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

-MacroNest: Configure maximum macro nesting

Description

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

Syntax

```
-MacroNest<Value>
```

Group

Language

Scope

Assembly Unit

Arguments

`<Value>`: max. allowed nesting level

Default

`3000`

Example

See the description of message A1004 for an example.

See also

Message A1004 (available in the Online Help)

-N: Display notify box

Description

Makes the Assembler display an alert box if there was an error during assembling. This is useful when running a makefile (see the *Build Tools* manual) because the Assembler waits for the user to acknowledge the message, thus suspending makefile processing. (The `N` stands for Notify.)

This feature is useful for halting and aborting a build using the Make Utility.

Syntax

```
-N
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-N
```

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

-NoBeep: No beep in case of an error

Description

Normally there is a ‘beep’ notification at the end of processing if there was an error. To have a silent error behavior, this ‘beep’ may be switched off using this option.

Syntax

```
-NoBeep
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-NoBeep
```

-NoDebugInfo: No debug information for ELF/DWARF files

Description

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

Syntax

```
-NoDebugInfo
```

Group

Language

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-NoDebugInfo
```

-NoEnv: Do not use environment

Description

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

When this option is given, the application does not use any environment (`default.env`, `project.ini` or tips file).

Syntax

```
-NoEnv
```

Group

Startup (This option cannot be specified interactively.)

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
xx.exe -NoEnv
```

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

See also

Environment

-ObjN: Object filename specification

Description

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

Syntax

```
-ObjN<FileName>
```

Group

Output

Scope

Assembly Unit

Arguments

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

Default

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

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

Example

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

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

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

See also

OBJPATH: Object file path

-Prod: Specify project file at startup

Description

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

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

Syntax

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

Group

None (This option cannot be specified interactively.)

Scope

None

Arguments

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

Default

None

Example

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

(Use the Assembler’s executable name instead of “`assembler`”.)

See also

Environment

-Struct: Support for structured types

Description

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

Syntax

```
-Struct
```

Group

Input

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-Struct
```

See also

Mixed C and Assembler Applications

-V: Prints the assembler version

Description

Prints the Assembler version and the current directory.

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

Syntax

```
-V
```

Group

Various

Scope

None

Arguments

None

Default

None

Example

-V produces the listing shown in Listing 5.28.

Listing 5.28 Example of a Version Listing

```
Command Line '-v'
Assembler V-5.0.8, Jul  7 2005
Directory: C:\Freescale\demo

Common Module V-5.0.7, Date Jul  7 2005
User Interface Module, V-5.0.17, Date Jul  7 2005
Assembler Kernel, V-5.0.13, Date Jul  7 2005
Assembler Target, V-5.0.8, Date Jul  7 2005
```

-View: Application standard occurrence

Description

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

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

Syntax

```
-View<kind>
```

Group

Host

Scope

Assembly Unit

Arguments

`<kind>` is one of the following:

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

Default

Application is started with arguments: `Minimized`.

Application is started without arguments: `Window`.

Example

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

-W1: No information messages

Description

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

Syntax

```
-W1
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-W1
```

-W2: No information and warning messages

Description

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

Syntax

```
-W2
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

None

Example

```
ASMOPTIONS=-W2
```

-WErrFile: Create "err.log" error file

Description

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

Syntax

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

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

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

Example

- `-WErrFileOn`

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

  existing `err.log` is not modified.

See also

-WStdout: Write to standard output

-WOutFile: Create error listing file

-Wmsg8x3: Cut filenames in Microsoft format to 8.3

Description

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

Syntax

`-Wmsg8x3`

Group

Messages

Scope

Assembly Unit

Default

None

Example

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

With the -Wmsg8x3 option set, the above message will be
x:\mysource.c(3): INFORMATION C2901: Unrolling loop
```

See also

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

-WmsgCE: RGB color for error messages

Description

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

Syntax

```
-WmsgCE<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCE255` changes the error messages to blue.

-WmsgCF: RGB color for fatal messages

Description

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

Syntax

```
-WmsgCF<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCF255` changes the fatal messages to blue.

-WmsgCI: RGB color for information messages

Description

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

Syntax

```
-WmsgCI<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCI255` changes the information messages to blue.

-WmsgCU: RGB color for user messages

Description

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

Syntax

```
-WmsgCU<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCU255` changes the user messages to blue.

-WmsgCW: RGB color for warning messages

Description

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

Syntax

```
-WmsgCW<RGB>
```

Group

Messages

Scope

Compilation Unit

Arguments

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

Default

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

Example

`-WmsgCW0` changes the warning messages to black.

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

Description

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

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

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

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

Syntax

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

Group

Messages

Scope

Assembly Unit

Arguments

`v`: Verbose format.

`m`: Microsoft format.

Default

```
-WmsgFbm
```

Example

Assume that the assembly source code in Listing 5.29 is to be assembled in the batch mode.

Listing 5.29 Example Assembly Source Code

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

The Assembler generates the error output (Listing 5.30) in the assembler window if it is running in batch mode:

Listing 5.30 Example Error Listing in the Microsoft (default) Format for Batch Mode

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

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

Listing 5.31 Example Error Listing in the Verbose Format for Batch Mode

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

See also

ERRORFILE: Filename specification error

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

-WmsgFob: Message format for batch mode

-WmsgFoi: Message format for interactive mode

-WmsgFonf: Message format for no file information

-WmsgFonp: Message format for no position information

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

Description

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

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

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

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

Syntax

`-WmsgFi[v|m]`

Group

Messages

Scope

Assembly Unit

Arguments

`v`: Verbose format.

`m`: Microsoft format.

Default

`-WmsgFiv`

Example

If the Assembler is running in interactive mode, the default error output is shown in the assembler window as in Listing 5.33.

Listing 5.32 Example Error Listing in the Default Mode for Interactive Mode

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

Setting the format to Microsoft, less information is displayed:

Listing 5.33 Example Error Listing in MIcrosoft Format for Interactive Mode

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

See also

ERRORFILE: Filename specification error

Assembler options:

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

-WmsgFob: Message format for batch mode

Description

With this option it is possible to modify the default message format in the batch mode. The formats in Listing 5.34 are supported (assume that the source file is `x:\Freescale\sourcefile.asmx`):

Listing 5.34 Supported Formats for Messages in Batch Mode

```
Format  Description  Example

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

Syntax

```
-WmsgFob<string>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

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

produces a message displayed in Listing 5.35 using the format in Listing 5.34. The options are set for producing the path of a file with its filename, extension, and line.

Listing 5.35 Error Message

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

See also

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

-WmsgFoi: Message format for interactive mode

Description

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

Listing 5.36 Supported Message Formats - Interactive Mode

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

Syntax

```
-WmsgFoi<string>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

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

produces a message in following format (Listing 5.37):

Listing 5.37 Error Message Resulting from the Statement Above

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

See also

ERRORFILE: Filename specification error

Assembler options:

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

-WmsgFonf: Message format for no file information

Description

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

Listing 5.38 Supported Formats

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

Syntax

`-WmsgFonf<string>`

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

`ASMOPTIONS=-WmsgFonf”%k %d: %m\n”`

produces a message in the following format:

`information L10324: Linking successful`

See also

ERRORFILE: Filename specification error

Assembler options:

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

-WmsgFonp: Message format for no position information

Description

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

Listing 5.39 Supported Message Formats when No Position Information is Available

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

Syntax

```
-WmsgFonp<string>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

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

Example

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

produces a message in following format:

```
information L10324: Linking successful
```

See also

ERRORFILE: Filename specification error

Assembler options:

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

-WmsgNe: Number of error messages

Description

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

Syntax

```
-WmsgNe<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

`50`

Example

```
ASMOPTIONS=-WmsgNe2
```

The Assembler stops assembling after two error messages.

See also

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

-WmsgNi: Number of information messages

Description

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

Syntax

```
-WmsgNi<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

```
50
```

Example

```
ASMOPTIONS=-WmsgNi10
```

Only ten information messages are logged.

See also

Assembler options:

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

-WmsgNu: Disable user messages

Description

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

`a`: This argument disables application-provided information about all included files.

`b`: This argument disables messages about reading files (e.g., the files used as input).

`c`: This argument disables messages about generated files.

`d`: This argument disables application-provided statistical information (e.g., code size, RAM/ROM usage) at the end of assembly.

`e`: This argument disables informal messages (e.g., memory model, floating point format).

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

Syntax

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

Group

Messages

Scope

None

Arguments

`a`: Disable messages about include files

`b`: Disable messages about reading files

`c`: Disable messages about generated files

`d`: Disable messages about processing statistics

`e`: Disable informal messages

Default

None

Example

```
-WmsgNu=c
```

-WmsgNw: Number of warning messages

Description

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

Syntax

```
-WmsgNw<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

`50`

Example

```
ASMOPTIONS=-WmsgNw15
```

Only 15 warning messages are logged.

See also

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

-WmsgSd: Setting a message to disable

Description

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

Syntax

```
-WmsgSd<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

```
-WmsgSd1801
```

See also

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

-WmsgSe: Setting a message to Error

Description

Allows changing a message to an error message.

Syntax

```
-WmsgSe<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

```
-WmsgSe1853
```

See also

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

-WmsgSi: Setting a message to Information

Description

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

Syntax

```
-WmsgSi<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

```
-WmsgSi1853
```

See also

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to Error
- -WmsgSw: Setting a message to Warning

-WmsgSw: Setting a message to Warning

Description

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

Syntax

```
-WmsgSw<number>
```

Group

Messages

Scope

Assembly Unit

Arguments

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

Default

None

Example

```
-WmsgSw2901
```

See also

- -WmsgSd: Setting a message to disable
- -WmsgSe: Setting a message to Error
- -WmsgSi: Setting a message to Information

-WOutFile: Create error listing file

Description

This option controls if a error listing file should be created at all. The error listing file contains a list of all messages and errors which are created during an assembly process. Since the text error feedback can also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the listing file is controlled by the environment variable ERRORFILE: Filename specification error.

Syntax

```
-WOutFile(On|Off)
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

Error listing file is created.

Example

```
-WOutFileOn
```

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

```
-WErrFileOff
```

No error file is created.

See also

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

-WStdout: Write to standard output

Description

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

Syntax

```
-WStdout(On|Off)
```

Group

Messages

Scope

Assembly Unit

Arguments

None

Default

output is written to `stdout`

Example

```
-WStdoutOn
```

All messages are written to `stdout`.

```
-WErrFileOff
```

Nothing is written to `stdout`.

See also

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

6

Sections

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

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

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

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

Section Attributes

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

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

Code Sections

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

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

Constant Sections

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

It is recommended that you define separate sections for the definition of variables and constant variables. This prevents any problems in the initialization of constant variables.

Data Sections

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

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

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

Section Types

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

This section covers these two types of sections:

- Absolute Sections
- Relocatable Sections

Absolute Sections

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

Listing 6.1 Example Source Code using ORG for Absolute Sections

```
          XDEF   entry

          ORG    $FC1000 LOGICAL
cst1:     DC.B   $A6
```

```
cst2:      DC.B   $BC

           ORG    $FC1100 LOGICAL
var:       DS.W   1

           ORG    $FC1200 LOGICAL
entry:
           LDL    R2, #%XGATE_8(cst1)
           LDH    R2, #%XGATE_8_H(cst1)
           LDB    R4, (R2, R0)

           LDL    R2, #%XGATE_8(cst2)
           LDH    R2, #%XGATE_8_H(cst2)
           LDB    R5, (R2, R0)

           ADD    R4, R4, R5

           LDL    R2, #%XGATE_8(cst2)
           LDH    R2, #%XGATE_8_H(cst2)
           STB    R4, (R2, R0)

loop:      BRA    loop
```

In the previous example, two bytes of storage are allocated starting at address $FC1000. The *constant* variable - cst1 - will be allocated one byte at address $FC1000 and another constant - cst2 - will be allocated one byte at address $FC1001. All subsequent instructions or data allocation directives will be located in this absolute section until another section is specified using the ORG or SECTION directives.

When using absolute sections, it is the user's responsibility to ensure that there is no overlap between the different absolute sections defined in the application. In the previous example, the programmer should ensure that the size of the section starting at address $F01A00 is not bigger than $200 bytes, otherwise the section starting at $F01A00 and the section starting at $F01C00 will overlap.

Applications containing only absolute sections must be linked. In that case, there should not be any overlap between the address ranges from the absolute sections defined in the assembly file and the address ranges defined in the linker parameter (PRM) file.

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

Listing 6.2 Example PRM File for Listing 6.1

```
LINK org01.abs       /* Name of the executable file generated. */
NAMES
     org01.o         /* Name of the object files in the application. */
END
```

```
SECTIONS
/* READ_ONLY memory area. There should be no overlap between this
memory area and the absolute sections defined in the assembly source
file. */
  MY_ROM  = READ_ONLY  0x4000 TO 0x4FFF ALIGN 2;

/* 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 ALIGN 2;

/* STACK memory area. There should be no overlap between this memory
area and the absolute sections defined in the assembly source file. */
  MY_STK  = READ_WRITE 0x3000 TO 0x3FFF; /* READ_WRITE memory area. */
END
PLACEMENT
  DEFAULT_RAM INTO MY_RAM; /* Variables are allocated in MY_RAM */
  DEFAULT_ROM INTO MY_ROM; /* Code is allocated in MY_ROM */
  SSTACK      INTO MY_STK; /* Stack is allocated in MY_STK.*/
END
INIT   entry        /* Entry is the entry point to the application. */
```

The linker PRM file contains at least:

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

Relocatable Sections

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

Listing 6.3 Example source code using SECTION for relocatable sections

```
          XDEF entry

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

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

codeSec:  SECTION ; Relocatable code section
entry:

          LDL    R2, #%XGATE_8(cst1)
          LDH    R2, #%XGATE_8_H(cst1)
          LDB    R4, (R2, R0)

          LDL    R2, #%XGATE_8(cst2)
          LDH    R2, #%XGATE_8_H(cst2)
          LDB    R5, (R2, R0)

          ADD    R4, R4, R5
          LDL    R2, #%XGATE_8(var)
          LDH    R2, #%XGATE_8_H(var)
          LDB    R4, (R2, R0)

loop:     BRA    loop
```

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

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

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

Example: Defining One RAM and One ROM Area

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

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

```
LINK test.abs /* Name of the executable file generated */
NAMES
     test.o   /* Name of the object files in the application */
END
SECTIONS
/* READ_ONLY memory area  */
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF ALIGN 2;
/* READ_WRITE memory area */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF ALIGN 2;
/* STACK memory area  */
  MY_STK  = READ_WRITE 0x3000 TO 0x3FFF;
END
PLACEMENT
  DEFAULT_RAM                INTO MY_RAM;
  DEFAULT_ROM, constSec      INTO MY_ROM;
  SSTACK                     INTO MY_STK;
END
INIT entry               /* Application Entry Point */
```

The linker PRM file contains at least:

- The name of the absolute file (LINK command).
- The name of the object files which should be linked (NAMES command).
- The specification of a memory area where the sections containing variables must be allocated. At least the predefined DEFAULT_RAM section must be placed there (SECTIONS and PLACEMENT commands).
- The specification of a memory area where the sections containing code or constants must be allocated. At least, the predefined DEFAULT_ROM section must be placed there (SECTIONS and PLACEMENT commands).
- The specification of the application entry point (INIT command).

According to the PRM file above:

- The dataSec section will be allocated starting at 0x2800.
- The codeSec section will be allocated starting at 0x2B00.
- The constSec section will be allocated next to the codeSec section.

Example: Defining Multiple RAM and ROM Areas

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

Listing 6.5 PRM File for Listing 6.3 Defining Multiple RAM and ROM Areas

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SECTIONS
  ROM_AREA_1= READ_ONLY  0x2B00 TO 0x2B7F ALIGN 2;/* READ_ONLY  mem. */
  ROM_AREA_2= READ_ONLY  0x2C00 TO 0x2C7F ALIGN 2;/* READ_ONLY  mem. */
  RAM_AREA_1= READ_WRITE 0x2800 TO 0x287F ALIGN 2;/* READ_WRITE mem. */
  RAM_AREA_2= READ_WRITE 0x2900 TO 0x297F ALIGN 2;/* READ_WRITE mem. */
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  dataSec                INTO RAM_AREA_2;
  DEFAULT_RAM            INTO RAM_AREA_1;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  constSec               INTO ROM_AREA_2;
  codeSec, DEFAULT_ROM   INTO ROM_AREA_1;
END
INIT entry               /*  Application entry point. */
```

The linker PRM file contains at least:

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

According to the PRM file in Listing 6.5,

- The dataSec section is allocated starting at 0x2900.
- The constSec section is allocated starting at 0x2C00.
- The codeSec section is allocated starting at 0x2B00.

Relocatable vs. Absolute Sections

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

Modularity

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

Multiple Developers

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

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

Early Development

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

Enhanced Portability

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

Tracking Overlaps

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

Reusability

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

7

Assembler Syntax

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

- Comment Line or
- Source Line.

Comment Line

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

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

Listing 7.1 Examples of comments

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

... (non comments)
```

Source Line

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

- Label Field,
- Operation Field,
- One or several operands, or
- A comment.

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

Label Field

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

NOTE For compatibility with other Assembler vendors, an identifier starting on column 1 is considered to be a label, even when it is not terminated by a colon.

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

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

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

Operation Field

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

- an XGATE instruction
- an XGATE instruction alias
- a directive name
- a Macro name.

XGATE Instructions

The table below lists all XGATE instructions. In addition to these instructions, some aliases are defined which should make assembly programs easier to read. The mnemonics of known aliases are shown with the associated plain XGATE instruction.

The mnemonics in the table below are sorted alphabetically. The symbolic names in the sample instructions are assumed to be assembly labels.

XGATE Instruction Set

Table 7.1 presents an overview of the available instructions:

Table 7.1 XGATE Instruction Set

Instruction	Description
ADC	Add with Carry
ADD	Add without Carry
ADDH	Add Immediate 8-Bit Constant (High Byte)
ADDL	Add Immediate 8-Bit Constant (Low Byte)
AND	Logical AND
ANDH	Logical AND Immediate 8-Bit Constant (High Byte)
ANDL	Logical AND Immediate 8-Bit Constant (Low Byte)
ASR	Arithmetic Shift Right
BCC	Branch if Carry Cleared (Same as BHS)
BCS	Branch if Carry Set (Same as BLO)
BEQ	Branch if Equal
BFEXT	Bit Field Extract
BFFO	Bit Field Find First One
BFINS	Bit Field Insert
BFINSI	Bit Field Insert and Invert
BFINSX	Bit Field Insert and XNOR
BGE	Branch if Greater than or Equal to Zero
BGT	Branch if Greater than Zero
BHI	Branch if Higher
BITH	Bit Test Immediate 8-Bit Constant (High Byte)
BITL	Bit Test Immediate 8-Bit Constant (Low Byte)
BLE	Branch if Less or Equal to Zero

Table 7.1 XGATE Instruction Set

Instruction	Description
BLS	Branch if Lower or Same
BLT	Branch if Lower than Zero
BMI	Branch is Minus
BNE	Branch if Not Equal
BPL	Branch if Plus
BRA	Branch Always
BRK	Break
BVC	Branch if Overflow Cleared
BVS	Branch if Overflow Set
CMPL	Compare Immediate 8-Bit Constant (Low Byte)
CPCH	Compare Immediate 8-Bit Constant with Carry (High Byte)
CSEM	Clear Semaphore
CSL	Logical Shift Left with Carry
CSR	Logical Shift Right with Carry
JAL	Jump and Link
LDB	Load Byte from Memory (Low Byte)
LDH	Load Immediate 8-Bit Constant (High Byte)
LDL	Load Immediate 8-Bit Constant (Low Byte)
LDW	Load Word from Memory
LSL	Logical Shift Left
LSR	Logical Shift Right
NOP	No Operation
OR	Logical OR
ORH	Logical OR Immediate 8-Bit Constant (High Byte)
ORL	Logical OR Immediate 8-Bit Constant (Low Byte)

Table 7.1 XGATE Instruction Set

Instruction	Description
PAR	Calculate Parity
ROL	Rotate Left
ROR	Rotate Right
RTS	Return to Scheduler
SBC	Subtract with Carry
SSEM	Set Semaphore
SEX	Sign Extent Byte to Word
SIF	Set Interrupt Flag
STB	Store Byte to Memory (Low Byte)
STW	Store Word to Memory
SUB	Subtract without Carry
SUBH	Subtract Immediate 8-Bit Constant (High Byte)
SUBL	Subtract Immediate 8-Bit Constant (Low Byte)
TFR	Transfer from and to Special Registers
XNOR	Logical Exclusive NOR
XNORH	Logical Exclusive NOR Immediate 8-Bit Constant (High Byte)
XNORL	Logical Exclusive NOR Immediate 8-Bit Constant (Low Byte)

XGATE Instruction Aliases

Table 7.2 XGATE Instruction Aliases

Instruction	Associated Alias	Description
BHS	BCC	Branch If Higher or Same
BLO	BCS	Branch if Carry Set
CLR RD	AND RD, R0, R0	Clear Register
CMP RS1, RS2	SUB R0, RS1, RS2	Compare
COM RD, RS	XNOR RS, R0, RD	One’s Complement
COM RD	XNOR RD, R0, RD	One’s Complement
CPC RS1, RS2	SBC R0, RS1, RS2	Compare with Carry
DEC RD	SUBL RD, #1	Decrement
INC RD	ADDL RD, #1	Increment
MOV RD, RS	OR RD, R0, RS	Move Register Content
NEG RD, RS	SUB RD, R0, RS	Two’s Complement
NEG RD	SUB RD, R0, RD	Two’s Complement
TST RS	SUB R0, RS, R0	Test Register

Directive

Assembler directives are described in the Assembler Directives chapter of this manual.

Macro

A user-defined macro can be invoked in the assembler source program. This results in the expansion of the code defined in the macro. Defining and using macros are described in the “Macros” chapter in this manual.

Operand Field: Addressing Modes

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

The following addressing mode notations are allowed in the operand field (Table 7.3):

Table 7.3 XGATE Addressing Mode Notation

Addressing Mode	Notation
Inherent	No operands
Immediate	#<imm>
Relative	<PC relative>
Register	<R>
Index Register plus Immediate Offset	(<R>,#<imm5>)
Index Register plus Register Offset	(<R>,<R>)
Index Register plus Register Offset with Post-increment	(<R>,<R>+)
Index Register plus Register Offset with Pre-decrement	(<R>,- <R>)

In the table above:

- <R> stands for a register
- #<imm> stands for an immediate expression
- #<imm5> stands for an immediate 5-bit expression

Inherent

Instructions using this addressing mode have no operands (Listing 7.2). The CPU does not need to perform memory access to complete the instruction.

Listing 7.2 Instructions with no operand

```
NOP
RTS
```

Immediate

The opcode contains the value to use with the instruction rather than the address of this value.

The effective address of the instruction is specified using the # character as in Listing 7.3.

Listing 7.3 Immediate addressing mode

```
loop:      LDL R4, #64
           BRA  loop
```

In this example, the hexadecimal value $64 is loaded in register R4. The size of the immediate operand is implied by the instruction context. In the case of a LDL the size of the immediate operand is 8-Bit.

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

Listing 7.4 Referring to address of symbol

```
           ORG $80 LOGICAL
var1:      DC.B $45, $67
           ORG $800 LOGICAL
main:
           LDL R2, #%XGATE_8(var1)
           LDH R2, #%XGATE_8_H(var1)
           LDB R4, (R2, R0)
           BRA  main
```

In this example, the address of the variable ‘var1’ ($80) is loaded in register R4.

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

For example, the following will load register R2 with the value stored at address $60.

```
LDL R2, $60
```

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.

See Listing 7.5 for an example of using the relative addressing mode.

Listing 7.5 Relative Addressing Mode

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

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

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

Listing 7.6 Example Specifying an Offset

```
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 (i.e. on label ‘main’).

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

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

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

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

Register

In this addressing mode registers are used as operands. The number of registers as operands is implied by the instruction context, as shown in Listing 7.7.

Listing 7.7 Example of Registers as Operands

```
        JAL R1
        SIF R2

        LSL R4, R5
        LSR R4, R5

        ADC R5, R6, R7
        SUB R5, R6, R7
```

In this example, JAL and SIF are used monadic; LSL and LSR are used dyadic; ADC and SUB are used triadic.

Index Register plus Immediate Offset

This addressing mode adds a 5-bit unsigned 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 [0...31].

This addressing mode may be used to access elements in an n-element table, where size is smaller than 32 bytes; as shown in Listing 7.8.

Listing 7.8 Example of Index Register plus Immediate Offset

```
            ORG $FD1000 LOGICAL
cst_tbl:    DC.B $5, $10, $18, $20, $28, $30
data_tbl:   DS.B 10
main:
            LDL R2, #%XGATE_8(cst_tbl)
            LDH R2, #%XGATE_8_H(cst_tbl)
            LDB R4, (R2, #1)

            LDL R2, #%XGATE_8(data_tbl)
            LDH R2, #%XGATE_8_H(data_tbl)
            STB R4, (R2, #2)
```

Index Register plus Register Offset

This addressing mode adds register to the base index register to form the memory address, which is referenced in the instruction.

This addressing mode may be used to access elements in an n-element table, where size is smaller than 65536 bytes; as shown in Listing 7.9

Listing 7.9 Example of Index Register plus Register Offset

```
            ORG $FC1000 LOGICAL
cst_tbl:    DC.B $5, $10, $18, $20, $28, $30
data_tbl:   DS.B 10
main:
            LDL R2, #%XGATE_8(cst_tbl)
            LDH R2, #%XGATE_8_H(cst_tbl)
            LDB R4, (R2, R0)
            LDL R2, #%XGATE_8(data_tbl)
            LDH R2, #%XGATE_8_H(data_tbl)
            LDL R3, #1
            STB R4, (R2, R3)
```

Index Register plus Register Offset with Post-Increment

This addressing mode adds register to the base index register to form the memory address, which is referenced in the instruction. After the register is added to the base index register it is incremented by 1.

This addressing mode may be used to access elements in an n-element table, where size is smaller than 65536 bytes; as shown in Listing 7.10

Listing 7.10 Example of Index Register plus Register Offset with Post-Increment

```
                   ORG $FB1000 LOGICAL
cst_tbl:    DC.B $5, $10, $18, $20, $28, $30
data_tbl:   DS.B 10
main:
            LDL R2, #%XGATE_8(cst_tbl)
            LDH R2, #%XGATE_8_H(cst_tbl)
            LDL R3, #1
            LDB R4, (R2, R3+)
            LDL R2, #%XGATE_8(data_tbl)
            LDH R2, #%XGATE_8_H(data_tbl)
            STB R4, (R2, R3+)
```

Index Register plus Register Offset with Pre-Decrement

This addressing mode adds register to the base index register to form the memory address, which is referenced in the instruction. Before the register is added to the base index register it is decremented by 1.

This addressing mode may be used to access elements in an n-element table, where size is smaller than 65536 bytes; as shown in Listing 7.11

Listing 7.11 Example of Index Register plus Register Offset with Pre-Decrement

```
            ORG $FA1000 LOGICAL
cst_tbl:    DC.B $5, $10, $18, $20, $28, $30
data_tbl:   DS.B 10
main:
            LDL R2, #%XGATE_8(cst_tbl)
            LDH R2, #%XGATE_8_H(cst_tbl)
            LDL R3, #1
            LDB R4, (R2, -R3)

            LDL R2, #%XGATE_8(data_tbl)
            LDH R2, #%XGATE_8_H(data_tbl)
            STB R4, (R2, -R3)
```

Fixup Types

The large integral operands like branch destination and 5/8-bit immediate operands may contain a fixup. The fixup type may be given explicitly by the programmer by using a syntax like:

```
LDL R2, #%XGATE_8(data_tbl)
```

The list of fixup types known to the assembler is given in the table below.

Table 7.4 Fixup Types

Fixup Symbol	Description
LOGICAL_8	low 8-bit (logical address space)
LOGICAL_8_H	high 8-bit (logical address space)
LOGICAL_16	16-bit (logical address space)
LOGICAL_32	32-bit (logical address space)
GLOBAL_8	low 8-bit (global address space)
GLOBAL_8_H	high 8-bit (global address space)
GLOBAL_16	16-bit (global address space)
GLOBAL_32	32-bit (global address space)
XGATE_8	low 8-bit (XGATE address space)
XGATE_8_H	high 8-bit (XGATE address space)
XGATE_16	16-bit (XGATE address space)

Comment Field

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

Example:

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

Symbols

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

- User-Defined Symbols
- External Symbols
- Undefined Symbols
- Reserved Symbols

User-Defined Symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

- The section, in which the memory location is defined
- The offset from the beginning of that section.

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION - Declare relocatable section assembler directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line (Listing 7.12).

Listing 7.12 Example of a User-Defined Relocatable SECTION

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

It is also possible to define a label with either an absolute or a previously defined relocatable value, using the SET - Set symbol value or EQU - Equate symbol value assembler directives.

Symbols with absolute values must be defined with constant expressions.

Listing 7.13 Example of a User-Defined Absolute and Relocatable SECTION

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

External Symbols

A symbol may be made external using the XDEF - External symbol definition assembler directive. In another source file, an XREF - External symbol reference assembler directive must reference it. Since its address is unknown in the referencing file, it is considered to be relocatable. See Listing 7.14 for an example of using `XDEF` and `XREF`.

Listing 7.14 Examples of External Symbols

```
      XREF extLabel        ; symbol defined in an other module.
                           ; extLabel is imported in the current module
      XDEF label           ; symbol is made external for other modules
                           ; label is exported from the current module
constSec: SECTION
label:    DC.W 1, extLabel
```

Undefined Symbols

If a label is neither defined in the source file nor declared external using `XREF`, the Assembler considers it to be undefined and generates an error message. Listing 7.15 shows an example of an undeclared label.

Listing 7.15 Example of an Undeclared Label

```
      XDEF entry
cstSec:  SECTION     ; Define a constant relocatable section
var1:    DC.B 5      ; Assign 5 to the symbol var1
dataSec: SECTION     ; Define a data relocatable section
data:    DS.B 1      ; Define one byte variable in RAM
codeSec: SECTION     ; Define a code relocatable section

entry:
         LDL         R2, #%XGATE_8(var1)
         LDH         R2, #%XGATE_8_H(var1)
         LDB         R4, (R2, R0)
```

```
main:
         ADDL        R4, #1
         LDL         R2, #%XGATE_8(data)
         LDH         R2, #%XGATE_8_H(data)
         STB         R4, (R2, R0)
         BRA         entry2; <- Undeclared user defined symbol: entry2
```

Reserved Symbols

Reserved symbols cannot be used for user-defined symbols.

Register names are reserved identifiers.

For the XGATE processor, the following identifiers are reserved:

- Integer registers `R0...R7`
- Condition code register `CCR`

Constants

The Assembler supports integer and ASCII string constants.

Integer Constants

The Assembler supports four representations of integer constants:

- A decimal constant is defined by a sequence of decimal digits (0-9).

  Example: 5, 512, 1024
- A hexadecimal constant is defined by a dollar character (`$`) followed by a sequence of hexadecimal digits (0-9, a-f, A-F).

  Example: `$5`, `$200`, `$400`
- An octal constant is defined by the commercial at character (`@`) followed by a sequence of octal digits (0-7).

  Example: `@5`, `@1000`, `@2000`
- A binary constant is defined by a percent character followed by a sequence of binary digits (0-1)

  Example:

  `%101`, `%1000000000`, `%10000000000`

The default base for integer constant is initially decimal, but it can be changed using the BASE - Set number base assembler directive. When the default base is not decimal, decimal values cannot be represented, because they do not have a prefix character.

String Constants

A string constant is a series of printable characters enclosed in single (') or double quote ("). Double quotes are only allowed within strings delimited by single quotes. Single quotes are only allowed within strings delimited by double quotes. See Listing 7.16 for a variety of string constants.

Listing 7.16 String Constants

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

Floating-Point Constants

The Macro Assembler does not support floating-point constants.

Operators

Operators recognized by the Assembler in expressions are:

- Addition and Subtraction Operators (Binary)
- Multiplication, division and modulo operators (binary)
- Sign Operators (Unary)
- Shift Operators (Binary)
- Bitwise Operators (Binary)
- Logical Operators (Unary)
- Logical Operators (Binary)
- Relational Operators (Binary)
- HIGH Operator
- PAGE Operator
- Force Operator (Unary)

Addition and Subtraction Operators (Binary)

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

Syntax

```
Addition:    <operand> + <operand>
Subtraction: <operand> - <operand>
```

Description

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

Addition between two relocatable operands is not allowed.

Example

See Listing 7.17 for an example of addition and subtraction operators.

Listing 7.17 Addition and Subtraction Operators

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

Multiplication, division and modulo operators (binary)

The multiplication, division, and modulo operators are `*`, `/`, and `%`, respectively.

Syntax

```
Multiplication: <operand> * <operand>
Division:       <operand> / <operand>
Modulo:         <operand> % <operand>
```

Description

The `*` operator multiplies two operands, the `/` operator performs an integer division of the two operands and returns the quotient of the operation. The `%` operator performs an integer division of the two operands and returns the remainder of the operation

The operands can be any expression evaluating to an absolute expression. The second operand in a division or modulo operation cannot be zero.

Example

See Listing 7.18 for an example of the multiplication, division, and modulo operators.

Listing 7.18 Multiplication, Division, and Modulo Operators

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

Sign Operators (Unary)

The (unary) sign operators are + and - .

Syntax

```
Plus:  +<operand>
Minus: -<operand>
```

Description

The + operator does not change the operand, whereas the – operator changes the operand to its two’s complement. These operators are valid for absolute expression operands.

Example

See Listing 7.19 for an example of the unary sign operators.

Listing 7.19 Unary Sign Operators

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

Shift Operators (Binary)

The binary shift operators are << and >>.

Syntax

```
Shift left:  <operand> << <count>
Shift right: <operand> >> <count>
```

Description

The << operator shifts its left operand left by the number of bits specified in the right operand.

The >> operator shifts its left operand right by the number of bits specified in the right operand.

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

Example

See Listing 7.20 for an example of the binary shift operators.

Listing 7.20 Binary Shift Operators

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

Bitwise Operators (Binary)

The binary bitwise operators are `&`, `|`, and `^`.

Syntax

```
Bitwise AND:       <operand> & <operand>
Bitwise OR:        <operand> | <operand>
Bitwise XOR:       <operand> ^ <operand>
```

Description

The `&` operator performs an AND between the two operands on the bit level.

The | operator performs an OR between the two operands on the bit level.

The ^ operator performs an XOR between the two operands on the bit level.

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

Example

See Listing 7.21 for an example of the binary bitwise operators

Listing 7.21 Binary Bitwise Operators

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

Bitwise Operators (Unary)

The unary bitwise operator is ~.

Syntax

```
One’s complement: ~<operand>
```

Description

The ~ operator evaluates the one’s complement of the operand.

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

Example

See Listing 7.22 for an example of the unary bitwise operator.

Listing 7.22 Unary Bitwise Operator

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

Logical Operators (Unary)

The unary logical operator is `!`.

Syntax

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

Description

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

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

Example

See Listing 7.23 for an example of the unary logical operator.

Listing 7.23 Unary Logical Operator

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

Logical Operators (Binary)

Syntax

```
Logical AND:       <operand> && <operand>
Logical OR:        <operand> || <operand>
```

Description

An operand of a boolean operator is considered FALSE if it equals 0, and TRUE otherwise.

The && operator performs a logical AND between the two operands, and evaluates to 1 (TRUE) or 0 (FALSE). The second operand is only evaluated if the first operand evaluates to TRUE.

The || operator performs a logical OR between the two operands, and evaluates to 1 (TRUE) or 0 (FALSE). The second operand is only evaluated if the first operand evaluates to FALSE.

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

Example

```
$E && 3      ; = $1 (TRUE)
$1 || fun   ; = $1 (TRUE), even if fun cannot be evaluated
```

Relational Operators (Binary)

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

Syntax

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

Description

These operators compare two operands and return `1` if the condition is ‘true’ or `0` if the condition is ‘false’.

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

Example

See Listing 7.24 for an example of the binary relational operators

Listing 7.24 Binary Relational Operators

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

HIGH Operator

This operator is not available for XGATE.

LOW Operator

This operator is not available for XGATE.

PAGE Operator

This operator is not available for XGATE.

Force Operator (Unary)

This operator is not available for XGATE.

Operator Precedence

Operator precedence follows the rules for ANSI - C operators.

Expression

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User defined symbols
- External symbols
- The special symbol '*' represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In the following example, the asterisk represents the location counter at the beginning of the DC directive:

  ```
  DC.W  1, 2, *-2
  ```

Once a valid expression has been fully evaluated by the Assembler, it is reduced as one of the following type of expressions:

- Absolute Expression: The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus it is a constant.
- Simple Relocatable Expression: The expression evaluates to an absolute offset from the start of a single relocatable section.
- Complex relocatable expression: The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The Assembler does not support such expressions.

All valid user defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in XREF directives, which are assumed to be relocatable symbols.

Absolute Expression

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

See Listing 7.25 for an example of an absolute expression.

Listing 7.25 Absolute Expression

```
Base:  SET $100
Label: EQU Base * $5 + 3
```

Expressions involving the difference between two relocatable symbols defined in the same file and in the same section evaluate to an absolute expression. An expression as `label2-label1` can be translated as:

Listing 7.26 label2-label1: Difference Between Two Relocatable Symbols

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

This can be simplified to (Listing 7.27):

Listing 7.27 Simplified Result for the Difference Between Two Relocatable Symbols

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

Example

In the example in Listing 7.28, the expression "`tabEnd-tabBegin`" evaluates to an absolute expression and is assigned the value of the difference between the offset of `tabEnd` and `tabBegin` in the section `DataSec`.

Listing 7.28 Absolute expression relating the difference between two relocatable symbols

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

ConstSec: SECTION
label:    EQU tabEnd-tabBegin     ; Absolute expression

CodeSec:  SECTION
entry:    NOP
```

Simple Relocatable Expression

A simple relocatable expression results from an operation such as one of the following:

```
<relocatable expression> + <absolute expression>
<relocatable expression> - <absolute expression>
<absolute expression> + <relocatable expression>
```

Listing 7.29 Example of Relocatable Expression

```
          XREF XtrnLabel
DataSec:  SECTION
tabBegin: DS.B  5
tabEnd:   DS.B  1
CodeSec:  SECTION
entry:
label1:
        LDL         R2, #%XGATE_8(tabBegin)   ; Simple relo. expression
        LDH         R2, #%XGATE_8_H(tabBegin) ; Simple relo. expression
         LDB         R4, (R2, R0)
         BRA        * - 6                     ; Simple relo. expression
```

Unary Operation Result

Table 7.5 describes the type of an expression according to the operator in an unary operation:

Table 7.5 Expression Type Resulting from Operator and Operand Type

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

Binary Operations Result

Table 7.6 describes the type of an expression according to the left and right operators in a binary operation:

Table 7.6 Expression Type Resulting from Operator and Operands

Operator	Left Operand	Right Operand	Expression
-	absolute	absolute	absolute
-	relocatable	absolute	relocatable
-	absolute	relocatable	complex
-	relocatable	relocatable	absolute
+	absolute	absolute	absolute
+	relocatable	absolute	relocatable
+	absolute	relocatable	relocatable
+	relocatable	relocatable	complex
*, /, %, <<, >>, |, &, ^	absolute	absolute	absolute
*, /, %, <<, >>, |, &, ^	relocatable	absolute	complex
*, /, %, <<, >>, |, &, ^	absolute	relocatable	complex
*, /, %, <<, >>, |, &, ^	relocatable	relocatable	complex

The evaluation of the binary logical operands && and || does not always require the evaluation of the second operand. For example, the following code evaluates to TRUE, even though `(entry + entry)` is of type complex:

```
(3 == 3 || (entry + entry))
```

Translation Limits

The following limitations apply to the Macro Assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Include may be nested up to `50`.
- The maximum line length is `1023`.

8

Assembler Directives

There are different class of assembler directives. The following tables give an overview of the different directives and their classes:

Directive Overview

Section Definition Directives

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

Table 8.1 Directives for Defining Sections

Directive	Description
ORG - Set location counter	Define an absolute section
SECTION - Declare relocatable section	Define a relocatable section
OFFSET - Create absolute symbols	Define an offset section

Constant Definition Directives

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

Table 8.2 Directives for Defining Constants

Directive	Description
EQU - Equate symbol value	Assign a name to an expression (cannot be redefined)
SET - Set symbol value	Assign a name to an expression (can be redefined)

Data Allocation Directives

The directives in Table 8.3 are used to allocate variables.

Table 8.3 Directives for Allocating Variables

Directive	Description
DC - Define constant	Define a constant variable
DCB - Define constant block	Define a constant block
DS - Define space	Define storage for a variable

Symbol Linkage Directives

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

Table 8.4 Symbol Linkage Directives

Directive	Description
ABSENTRY - Application entry point	Specify the application entry point when an absolute file is generated
XDEF - External symbol definition	Make a symbol public (visible from outside)
XREF - External symbol reference	Import reference to an external symbol.

Assembly Control Directives

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

Table 8.5 Assembly Control Directives

Directive	Description
ALIGN - Align location counter	Define Alignment Constraint
BASE - Set number base	Specify default base for constant definition
END - End assembly	End of assembly unit
EVEN - Force word alignment	Define 2-byte alignment constraint
FAIL - Generate error message	Generate user defined error or warning messages
INCLUDE - Include text from another file	Include text from another file.
LONGEVEN - Forcing long-word alignment	Define 4 Byte alignment constraint

Listing-File Control Directives

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

Table 8.6 Listing-file Control Directives

Directive	Description
CLIST - List conditional assembly	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not.
LIST - Enable listing	Specify that all subsequent instructions must be inserted in the listing file.
LLEN - Set line length	Define line length in assembly listing file.
MLIST - List macro expansions	Specify if the macro expansions must be inserted in the listing file.
NOLIST - Disable listing	Specify that all subsequent instruction must not be inserted in the listing file.
NOPAGE - Disable paging	Disable paging in the assembly listing file.
PAGE - Insert page break	Insert page break.
PLEN - Set page length	Define page length in the assembler listing file.
SPC - Insert blank lines	Insert an empty line in the assembly listing file.
TABS - Set tab length	Define number of character to insert in the assembler listing file for a TAB character.
TITLE - Provide listing title	Define the user defined title for the assembler listing file.

Macro Control Directives

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

Table 8.7 Macro Control Directives

Directive	Description
ENDM - End macro definition	End of user defined macro.
MACRO - Begin macro definition	Start of user defined macro.
MEXIT - Terminate macro expansion	Exit from macro expansion.

Conditional Assembly Directives

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

Table 8.8 Conditional Assembly Directives

Directive	Description
ELSE - Conditional assembly	alternate block
IF - Conditional assembly	Start of conditional block. A boolean expression follows this directive.
IFcc - Conditional assembly	Test if two string expressions are equal.
IFDEF	Test if a symbol is defined.
IFEQ	Test if an expression is null.
IFGE	Test if an expression is greater or equal to 0.
IFGT	Test if an expression is greater than 0.
IFLE	Test if an expression is less or equal to 0.
IFLT	Test if an expression is less than 0.
IFNC	Test if two string expressions are different.
IFNDEF	Test if a symbol is undefined
IFNE	Test if an expression is not null.

Detailed Descriptions of All Assembler Directives

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

ABSENTRY - Application entry point

Description

This directive is used to specify the application Entry Point when the Assembler directly generates an absolute file. The `-FA2` assembly option - ELF/DWARF 2.0 Absolute File - must be enabled.

Using this directive, the entry point of the assembly application is written in the header of the generated absolute file. When this file is loaded in the debugger, the line where the entry point label is defined is highlighted in the source window.

This directive is ignored when the Assembler generates an object file.

NOTE This instruction only affects the loading on an application by a debugger. It tells the debugger which initial PC should be used. In order to start the application on a target - initialize the Reset vector.

Syntax

```
ABSENTRY <label>
```

Synonym

None

Example

If the example in Listing 8.1 is assembled using the -FA2 assembler option, an ELF/DWARF 2.0 Absolute file is generated.

Listing 8.1 Using ABSENTRY to Specify an Application Entry Point

```
       ABSENTRY entry

       ORG   $fffe
Reset: DC.W  entry
       ORG   $70
```

```
entry: NOP
       NOP
main:  RSP
       NOP
       BRA   main
```

According to the `ABSENTRY` directive, the entry point will be set to the address of entry in the header of the absolute file.

ALIGN - Align location counter

Description

This directive forces the next instruction to a boundary that is a multiple of <n>, relative to the start of the section. The value of `<n>` must be a positive number between 1 and 32767. The `ALIGN` directive can force alignment to any size. The filling bytes inserted for alignment purpose are initialized with `'\0'`.

`ALIGN` can be used in code or data sections.

Syntax

```
ALIGN <n>
```

Synonym

```
ALIGN.B (= ALIGN 1)
ALIGN.W (= ALIGN 2)
ALIGN.L (= ALIGN 4)
ALIGN.F (= ALIGN 4)
ALIGN.D (= ALIGN 8)
```

Example

The example shown in Listing 8.2 aligns the `HEX` label to a location, which is a multiple of 16 (in this case, location `00010 (Hex)`)

Listing 8.2 Aligning the HEX Label to a Location

```
Assembler

Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1
    2    2   000000 6869 6768          DC.B  "high"
    3    3   000004 0000 0000          ALIGN 16
             000008 0000 0000
             00000C 0000 0000
    4    4
    5    5
    6    6   000010 7F          HEX:    DC.B 127 ; HEX is allocated
    7    7                                        ; on an address,
    8    8                                        ; which is a
    9    9                                        ; multiple of 16.
```

BASE - Set number base

Description

The directive sets the default number base for constants to `<n>`. The operand `<n>` may be prefixed to indicate its number base; otherwise, the operand is considered to be in the current default base. Valid values of `<n>` are 2, 8, 10, 16. Unless a default base is specified using the `BASE` directive, the default number base is decimal.

Syntax

```
BASE <n>
```

Synonym

None

Example

See Listing 8.3 for examples of setting the number base.

Listing 8.3 Setting the Number Base

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

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

CLIST - List conditional assembly

Description

The `CLIST` directive controls the listing of subsequent conditional assembly blocks. It precedes the first directive of the conditional assembly block to which it applies, and remains effective until the next `CLIST` directive is read.

When the `ON` keyword is specified in a `CLIST` directive, the listing file includes all directives and instructions in the conditional assembly block, even those which do not generate code (which are skipped).

When the `OFF` keyword is entered, only the directives and instructions that generate code are listed.

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

Syntax

```
CLIST [ON|OFF]
```

Synonym

None

Example

Listing 8.4 is an example where the `CLIST OFF` option is used.

Listing 8.4 With CLIST OFF

```
        CLIST OFF
Try:    EQU     0
```

```
        IFEQ    Try
          DC.W   $12
        ELSE
          DC.W   $00
        ENDIF
```

Listing 8.5 is the corresponding listing file.

Listing 8.5 Example Assembler Listing Using CLIST OFF

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    2    2                      Try:    EQU     0
    3    3                              IFEQ    Try
    4    4   000000 0012                  DC.W   $12
    5    5                              ELSE
    7    7                              ENDIF
```

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

Listing 8.6 CLIST ON is Selected

```
        CLIST ON
Try:    EQU     0
        IFEQ    Try
          DC.W   $12
        ELSE
          DC.W   $00
        ENDIF
```

Listing 8.7 is the corresponding listing file.

Listing 8.7 Example Assembler Listing Using CLIST ON

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    2    2                      Try:    EQU     0
    3    3                              IFEQ    Try
    4    4   000000 0012                  DC.W   $12
    5    5                              ELSE
    6    6                                DC.W   $00
    7    7                              ENDIF
```

DC - Define constant

Description

The `DC` directive defines constants in memory. It can have one or more `<expression>` operands, which are separated by commas. The `<expression>` can contain an actual value (binary, octal, decimal, hexadecimal, or ASCII). Alternatively, the `<expression>` can be a symbol or expression that can be evaluated by the Assembler as an absolute or simple relocatable expression. One memory block is allocated and initialized for each expression.

The following rules apply to size specifications for `DC` directives:

- `DC.B:` One byte is allocated for numeric expressions. One byte is allocated per ASCII character for strings (Listing 8.8).
- `DC.W`: Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a two-byte boundary (Listing 8.9).
- `DC.L`: Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a four byte boundary (Listing 8.10
- `DC.F`: Four bytes are allocated for floating point values in IEEE32 format (Listing 8.12).
- `DC.D`: Eight bytes are allocated for floating point values in IEEE64 format (Listing 8.13).

Syntax

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

where <size> =

- `B` (default), `W` or `L` for integral values
- `F` or `D` for floating point values

Synonym

```
DCW (= 2 byte DCs),  DCL (= 4 byte DCs),
FCB (= DC.B), FDB    (= 2 byte DCs),
FQB (= 4 byte DCs)
```

Examples

Listing 8.8 Example for DC.B

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

Listing 8.9 Example for DC.W

```
000000 0041 4243   Label: DC.W "ABCDE"
000004 4445
000006 000A 000A          DC.W %1010, @12, 1, $A
00000A 0001 000A
00000E xxxx               DC.W Label
```

Listing 8.10 Example for DC.L

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445
000008 0000 000A          DC.L %1010, @12, 1, $A
00000C 0000 000A
000010 0000 0001
000014 0000 000A
000018 xxxx xxxx          DC.L Label
```

Listing 8.11 Example for DC.L

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445
000008 0A00 0000          DC.L %1010, @12, 1, $A
00000C 0A00 0000
000010 0100 0000
000014 0A00 0000
000018 xxxx xxxx          DC.L Label
```

Listing 8.12 Example for DC.F

```
000000 3F80 0000           DC.F    1
000004 3900 F990           DC.F    1.23e-4
```

Listing 8.13 Example for DC.D

```
000000 3FF0 0000          DC.D    1
000004 0000 0000
000008 3F20 1F31          DC.D    1.23e-4
00000C F46E D246
```

If the value in an operand expression exceeds the size of the operand, the value is truncated and a warning message is generated.

See also

- DCB - Define constant block
- DS - Define space
- ORG - Set location counter
- SECTION - Declare relocatable section

DCB - Define constant block

Description

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

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

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

The following rules apply to size specifications for `DCB` directives:

- `DCB.B`: One byte is allocated for numeric expressions.
- `DCB.W`: Two bytes are allocated for numeric expressions.
- `DCB.L`: Four bytes are allocated for numeric expressions.
- `DCB.F`: Four bytes are allocated for IEEE32 values.DCB.D: Eight bytes are allocated for IEEE64 values.

Syntax

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

where <size> =

- B (default), W or L for integral values
- F or D for floating point values

Examples

Listing 8.14 Examples of DCB Directives

```
000015 24FD A48A          DCB.F 3, 1.1e-16
000019 24FD A48A
00001D 24FD A48A
000021 3FF0 0000          DCB.D 2, 1.0
000025 0000 0000
000029 3FF0 0000
00002D 0000 0000
```

See also

- DC - Define constant
- DS - Define space
- ORG - Set location counter
- SECTION - Declare relocatable section

DS - Define space

Description

The `DS` directive is used to reserve memory for variables (Listing 8.15). The content of the memory reserved is not initialized. The length of the block is `<size> * <count>`.

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

Listing 8.15 Examples of DS Directives

```
Counter:  DS.B  2 ; 2 continuous bytes in memory
          DS.B  2 ; 2 continuous bytes in memory
                  ; can only be accessed through the label Counter
          DS.W  5 ; 5 continuous words in memory
```

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

NOTE Storage allocated with a `DS` directive may end up in constant data section or even in a code section, if the same section contains constants or code as well. The Assembler allocates only a complete section at once.

Syntax

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

where `<size>` = `B` (default), `W`, L, F, or D.

Synonym

`RMB (= DS.B)`

`RMD (2 bytes)`

`RMQ (4 bytes)`

Example

In Listing 8.16, a variable, a constant, and code were put in the same section. Because code has to be in ROM, then all three elements must be put into ROM. In order to allocate them separately, put them in different sections (Listing 8.17).

Listing 8.16 Poor Memory Allocation

```
; How it should NOT be done ...
Counter:        DS 1     ; 1-byte used
InitialCounter: DC.B $f5 ; constant $f5
main:           NOP      ; NOP instruction
```

Listing 8.17 Proper Memory Allocation

```
DataSect:      SECTION    ; separate section for variables
Counter:       DS 1       ; 1-byte used

ConstSect:     SECTION  ; separate section for constants
InitialCounter: DC.B $f5  ; constant $f5

CodeSect:      SECTION    ; section for code
main:           NOP       ; NOP instruction
```

An ORG directive also starts a new section.

See also

- DC - Define constant
- ORG - Set location counter
- SECTION - Declare relocatable section

ELSE - Conditional assembly

Description

If <condition> is true, the statements between IF and the corresponding ELSE directive are assembled (generate code).

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

Syntax

```
IF <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

```
ELSEC
```

Example

Listing 8.18 is an example of the use of conditional assembly directives:

Listing 8.18 Various Conditional Assembly Directives

```
Try:    EQU     0
        IF  Try != 0
          DC.W  $1234
        ELSE
          DC.W  $5678
        ENDIF
```

The value of *Try* determines the instruction to be assembled in the program. As shown, the "`DC.W $5678`" instruction is assembled. Changing the operand of the "`EQU`" directive to one causes the "`DC.W $1234`" instruction to be assembled instead. The following shows the listing provided by the assembler for these lines of code:

Listing 8.19 Output Listing of Listing 8.18

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      Try:    EQU     0
    2    2                              IF  Try != 0
    4    4                              ELSE
    5    5   000000 5678                  DC.W  $5678
    6    6                              ENDIF
```

END - End assembly

Description

The END directive indicates the end of the source code. Subsequent source statements in this file are ignored. The END directive in included files skips only subsequent source statements in this include file. The assembly continues in the including file in a regular way.

Syntax

```
END
```

Synonym

None

Example

The END statement in Listing 8.20 causes any source code after the END statement to be ignored, as in Listing 8.21.

Listing 8.20 Source File

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

Listing 8.21 Generated Listing File

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 1234        Label:  DC.W  $1234
    2    2   000002 5678                DC.W  $5678
```

ENDIF - End conditional assembly

Description

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

Syntax

```
ENDIF
```

Synonym

```
ENDC
```

Example

See Listing 8.25 in the IF section.

See also

IF - Conditional assembly assembler directive

ENDM - End macro definition

Description

The ENDM directive terminates the macro definition (Listing 8.22).

Syntax

```
ENDM
```

Synonym

None

Example

The following example (Listing 8.22) shows how to define and use macros LE.W and LE.L which reverse the byte order of integral constants.

Listing 8.22 Using ENDM to Terminate a Macro Definition

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      LE.W:   MACRO
   2    2                              DC.B (\1)&$FF
   3    3                              DC.B (\1)>>8
   4    4                              ENDM
   5    5                      LE.L:   MACRO
   6    6                              LE.W (\1)&$FFFF
   7    7                              LE.W (\1)>>16
   8    8                              ENDM
   9    9
  10   10   000000 1234 5678           DC.L $12345678
  11   11                              LE.L $12345678
  12    6m                    +        LE.W ($12345678)&$FFFF
  13    2m  000004 78         +        DC.B (($12345678)&$FFFF)&$FF
  14    3m  000005 56         +        DC.B (($12345678)&$FFFF)>>8
  15    7m                    +        LE.W ($12345678)>>16
  16    2m  000006 34         +        DC.B (($12345678)>>16)&$FF
  17    3m  000007 12         +        DC.B (($12345678)>>16)>>8
```

EQU - Equate symbol value

Description

The EQU directive assigns the value of the <expression> in the operand field to <label>. The <label> and <expression> fields are both required, and the <label> cannot be defined anywhere else in the program.

Syntax

```
<label>: EQU <expression>
```

Synonym

None

Example

See Listing 8.23 for examples of using the EQU directive.

Listing 8.23 Using EQU to set variables

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

                   Time:   DS.B  3
        0000 0000  Hour:   EQU   Time   ; first byte addr.
        0000 0002  Minute: EQU   Time+1 ; second byte addr
        0000 0004  Second: EQU   Time+2 ; third byte addr
```

EVEN - Force word alignment

Description

This directive forces the next instruction to the next even address relative to the start of the section. EVEN is an abbreviation for ALIGN 2. Some processors require word and long word operations to begin at even address boundaries. In such cases, the use of the EVEN directive ensures correct alignment. Omission of this directive can result in an error message.

Syntax

EVEN

Synonym

None

Example

See Listing 8.24 for instances where the EVEN directive causes padding bytes to be inserted.

Listing 8.24 Using the Force Word Alignment Directive

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000                   ds.b  4
    2    2                     ; location count has an even value
    3    3                     ; no padding byte inserted.
    4    4                            even
    5    5   000004                         ds.b   1
    6    6                            ; location count has an odd value
    7    7                            ; one padding byte inserted.
    8    8   000005                         even
    9    9   000006                         ds.b   3
   10   10                            ; location count has an odd value
   11   11                            ; one padding byte inserted.
   12   12   000009                         even
   13   13          0000 000A         aaa:   equ    10
```

See also

ALIGN - Align location counter assembly directive

FAIL - Generate error message

Description

There are three modes of the FAIL directive, depending upon the operand that is specified:

- If <arg> is a number in the range [0-499], the Assembler generates an error message, including the line number and argument of the directive. The Assembler does not generate an object file.
- If <arg> is a number in the range [500-$FFFFFFFF], the Assembler generates a warning message, including the line number and argument of the directive.
- If a string is supplied as an operand, the Assembler generates an error message, including the line number and the <string>. The Assembler does not generate an object file.
- The FAIL directive is primarily intended for use with conditional assembly to detect user-defined errors or warning conditions.

Syntax

```
FAIL  <arg>|<string>
```

Synonym

None

Example

To illustrate the `FAIL` directive, we define and use a macro `LE.B` to revert two bytes in a `DC.B` directive. We want this macro to issue an error if it is “called” with zero or more than three arguments. Furthermore, we expect a warning message if `LE.B` is used with a single byte.

```
LE.B:   MACRO
        IFC "\1",""
          FAIL  "no data"
          MEXIT
        ENDIF
        IFC "\2",""
          FAIL  600
          DC.B  \1
          MEXIT
        ENDIF
        IFNC "\3",""
          FAIL  400
        ENDIF
        DC.B  \2,\1
        ENDM

        LE.B
        LE.B    $12
        LE.B    $12,$34
        LE.B    $12,$34,$56
```

Assembling this code, we obtain the following error messages:

```
>> in "fail.asm", line 3, col 0, pos 50
          FAIL  "no data"
                ^
ERROR A2338: no data
INFORMATION Macro Expansion           FAIL  "no data"

>> in "fail.asm", line 7, col 0, pos 128
          FAIL  600
                  ^
WARNING A2332: FAIL found
INFORMATION Macro Expansion           FAIL  600

>> in "fail.asm", line 12, col 0, pos 218
          FAIL  400
                  ^
ERROR A2329: FAIL found
INFORMATION Macro Expansion           FAIL  400
```

IF - Conditional assembly

Description

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

The expected syntax for `<condition>` is:

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

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

Syntax

```
IF <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

`None`

Example

Listing 8.25 is an example of the use of conditional assembly directives.

Listing 8.25 IF and ENDIF

```
Try:    EQU     0
        IF Try != 0
          DC.W  $12
        ELSE
          DC.W  $34
        ENDIF
```

The value of *Try* determines the instruction to be assembled in the program. As shown, the "`DC.W $34`" instruction is assembled. Changing the operand of the

"EQU" directive to 1 would cause the "DC.W $12" instruction to be assembled instead. The following shows the listing provided by the assembler for these lines of code:

Listing 8.26 Output listing after conditional assembly

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1          0000 0000   Try:    EQU     0
   2    2          0000 0000           IF Try != 0
   4    4                              ELSE
   5    5   000000 0034                  DC.W  $34
   6    6                              ENDIF
```

IFcc - Conditional assembly

Description

These directives can be replaced by the `IF` directive `Ifcc <condition>` is true, the statements immediately following the `Ifcc` directive are assembled. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached, after which assembly moves to the statements following the `ENDIF` directive. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Table 8.9 lists the available conditional types:

Table 8.9 Conditional Assembly Types

Ifcc	Condition	Meaning
ifeq	<expression>	if <expression> == 0
ifne	<expression>	if <expression> != 0
iflt	<expression>	if <expression> < 0
ifle	<expression>	if <expression> <= 0
ifgt	<expression>	if <expression> > 0
ifge	<expression>	if <expression> >= 0
ifc	<string1>, <string2>	if <string1> == <string2>
ifnc	<string1>, <string2>	if <string1> != <string2>
ifdef	<label>	if <label> was defined
ifndef	<label>	if <label> was not defined

Syntax

```
IFcc <condition>
  [<assembly language statements>]
[ELSE]
  [<assembly language statements>]
ENDIF
```

Synonym

None

Example

Listing 8.27 illustrates the IFDEF conditional directive. The example shows how to make sure that the label ‘main’ is declared in an included file.

Listing 8.27 Using the IFNE Conditional Assembler Directive

```
INCLUDE "ifcc.inc"

IFDEF main
  DC.W main
ELSE
  FAIL "main not defined"
ENDIF
```

The following assembly listing shows what happens if the label ‘main’ is declared in the header file.

Listing 8.28 Output Listing for Listing 8.27

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                              INCLUDE "ifcc.inc"
   2    1i                             XREF main
   3    2
   4    3          0000 0001           IFDEF main
   5    4   000000 xxxx                  DC.W main
   6    5                              ELSE
   8    7                              ENDIF
```

INCLUDE - Include text from another file

Description

This directive causes the included file to be inserted in the source input stream. The `<file specification>` is not case-sensitive and must be enclosed in quotation marks.

The Assembler attempts to open `<file specification>` relative to the current working directory. If the file is not found there, then it is searched for relative to each path specified in the GENPATH: Search path for input file environment variable.

Syntax

```
INCLUDE <file specification>
```

Synonym

None

Example

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

LIST - Enable listing

Description

Specifies that instructions following this directive must be inserted into the listing and into the debug file. This is a default option. The listing file is only generated if the -L: Generate a listing file assembler option is specified on the command line.

The source text following the `LIST` directive is listed until a NOLIST - Disable listing or an END - End assembly assembler directive is reached

This directive is not written to the listing and debug files.

Syntax

```
LIST
```

Synonym

None

Example

The assembly source code using the `LIST` and `NOLIST` directives in Listing 8.29 generates the output listing in Listing 8.30.

Listing 8.29 Using the LIST and NOLIST Assembler Directives

```
aaa:     NOP

         LIST
bbb:     NOP
         NOP

         NOLIST
ccc:     NOP
         NOP

         LIST
ddd:     NOP          NOP
```

Listing 8.30 Output Listing Generated by Running Listing 8.29

```
 Abs. Rel.    Loc    Obj. code    Source line
 ---- ----   ------ ---------    -----------
    1    1   000000 0100         aaa:    NOP
    2    2
    4    4   000002 0100         bbb:    NOP
    5    5   000004 0100                 NOP
    6    6
   12   12   00000A 0100         ddd:    NOP
   13   13   00000C 0100                 NOP
```

LLEN - Set line length

Description

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

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

Syntax

```
LLEN<n>
```

Synonym

None

Example

The following portion of code in Listing 8.32 generates the listing file in Listing 8.32. Notice that the 'LLEN 24' directive causes the output at the location-counter line 7 to be truncated.

Listing 8.31 Example Assembly Source Code Using LLEN

```
          DC.B   $55
          LLEN   32
          DC.W   $1234, $4567
          LLEN   24
          DC.W   $1234, $4567
          EVEN
```

Listing 8.32 Formatted Assembly Output Listing as a Result of Using LLEN

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 55                  DC.B   $55
    2    2
    4    4   000001 1234 4567           DC.W   $1234, $4567
    5    5
    7    7   000005 1234 4567           DC.W   $1234, $
    8    8   000009 00                  EVEN
```

LONGEVEN - Forcing long-word alignment

Description

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

Syntax

```
LONGEVEN
```

Synonym

None

Example

See Listing 8.33 for an example where `LONGEVEN` aligns the next instruction to have its location counter to be a multiple of four (bytes).

Listing 8.33 Forcing Long-Word Alignment

```
    1    1   000000 55             DC.B    $55
    2    2   000001 0000 00        LONGEVEN
    3    3                         ;;  3 filling bytes required
    4    4   000004 0012 0034      DC.W    $12, $34
    5    5   000008                LONGEVEN
    6    6                         ;; no filling bytes required
    7    7   000008 0000 5678      DC.L    $5678
```

MACRO - Begin macro definition

Description

The `<label>` of the `MACRO` directive is the name by which the macro is called. This name must not be a processor machine instruction or assembler directive name. For more information on macros, see Macros.

Syntax

```
<label>: MACRO
```

Synonym

None

Example

See Listing 8.34 for a macro definition.

Listing 8.34 Example Macro Definition

```
LE.W:   MACRO
        DC.B (\1)&$FF
        DC.B (\1)>>8
        ENDM
LE.L:   MACRO
        LE.W (\1)&$FFFF
        LE.W (\1)>>16
        ENDM

        DC.L $12345678
        LE.L $12345678
```

MEXIT - Terminate macro expansion

Description

`MEXIT` is usually used together with conditional assembly within a macro. In that case it may happen that the macro expansion should terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the ENDM - End macro definition directive.

Syntax

`MEXIT`

Synonym

None

Example

The macro in Listing 8.35 allows the replication of simple instructions or directives and generates the listing file in Listing 8.36.

Listing 8.35 Macro Allowing Replication of Instructions or Directives

```
MANY:   MACRO
        ;; \1 = count
        ;; \2 = what
        IF (\1 <= 0)
          MEXIT
        ENDIF
        \2
        MANY (\1-1),\2
        ENDM
        MANY  2, DC.W "fun"
```

Listing 8.36 Generated Listing File

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      MANY:   MACRO
    2    2                              ;; \1 = count
    3    3                              ;; \2 = what
    4    4                              IF (\1 <= 0)
    5    5                                MEXIT
    6    6                              ENDIF
    7    7                              \2
    8    8                              MANY (\1-1),\2
    9    9                              ENDM
   10   10
   11   11                              MANY  2, DC.W "fun"
   12    2m                    +        ;; 2 = count
   13    3m                    +        ;; DC.W "fun" = what
   14    4m         0000 0000  +        IF (2 <= 0)
   16    6m                    +        ENDIF
   17    7m  000000 0066 6F6F  +        DC.W "fun"
   18    8m                    +        MANY (2-1),DC.W "fun"
   19    2m                    +        ;; (2-1) = count
   20    3m                    +        ;; DC.W "fun" = what
   21    4m         0000 0000  +        IF ((2-1) <= 0)
   23    6m                    +        ENDIF
   24    7m  000004 0066 6F6F  +        DC.W "fun"
   25    8m                    +        MANY ((2-1)-1),DC.W "fun"
   26    2m                    +        ;; ((2-1)-1) = count
   27    3m                    +        ;; DC.W "fun" = what
   28    4m         0000 0001  +        IF (((2-1)-1) <= 0)
   30    5m                    +          MEXIT
   31    6m                    +        ENDIF
   32    7m                    +        DC.W "fun"
   33    8m                    +        MANY (((2-1)-1)-1),DC.W "fun"
```

MLIST - List macro expansions

Description

When the ON keyword is entered with an MLIST directive, the Assembler includes the macro expansions in the listing and in the debug file.

When the OFF keyword is entered, the macro expansions are omitted from the listing and from the debug file.

This directive is not written to the listing and debug file, and the default value is ON.

Syntax

```
MLIST [ON|OFF]
```

Synonym

None

Example

Assembling the code in Listing 8.37 generates the listing file in Listing 8.38.

Listing 8.37 Sample Code

```
DC.PC:  MACRO
        DC.W    *-main
        ENDM

main:   SECTION
        DC.PC
        MLIST   OFF
        DC.PC
        MLIST   ON
        DC.PC
```

Listing 8.38 Generated Listing File

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      DC.PC:  MACRO
    2    2                              DC.W    *-main
    3    3                              ENDM
    4    4
    5    5                      main:   SECTION
    6    6                              DC.PC
    7    2m   000000 0000     +         DC.W     *-main
    9    8                              DC.PC
   12   10                              DC.PC
   13    2m   000004 0004     +         DC.W     *-main
```

The `MLIST` directive does not appear in the listing file. When a macro is called after a `MLIST ON`, it is expanded in the listing file. If the `MLIST OFF` is encountered before the macro call, the macro is not expanded in the listing file.

NOLIST - Disable listing

Description

Suppresses the printing of the following instructions in the assembly listing and debug file until a LIST - Enable listing assembler directive is reached.

Syntax

`NOLIST`

Synonym

`NOL`

Example

See Listing 8.39 for an example of using `LIST` and `NOLIST`.

Listing 8.39 Examples of LIST and NOLIST

```
aaa:   NOP

       LIST
bbb:   NOP
       NOP

       NOLIST
ccc:   NOP
       NOP

       LIST
ddd:   NOP
```

```
        NOP
```

The listing above generates the listing file in Listing 8.40.

Listing 8.40 Assembler Output Listing from the Assembler Source Code in Listing 8.39

```
XGATE-Assembler

 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1   000000 0100        aaa:    NOP
    2    2
    4    4   000002 0100        bbb:    NOP
    5    5   000004 0100                NOP
    6    6
   12   12   00000A 0100        ddd:    NOP
   13   13   00000C 0100                NOP
```

See also

LIST - Enable listing assembler directive

NOPAGE - Disable paging

Description

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

Syntax

```
NOPAGE
```

Synonym

None

OFFSET - Create absolute symbols

Description

The `OFFSET` directive declares an offset section and initializes the location counter to the value specified in `<expression>`. The `<expression>` must be absolute and may not contain references to external, undefined or forward defined labels.

Syntax

```
OFFSET <expression>
```

Synonym

None

Example

The following example shows how the OFFSET directive can be used to access an element of a structure.

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                              ;; "type definition"
   2    2                              OFFSET  0
   3    3  a000000                 id:     DS.B    1
   4    4  a000001                         EVEN
   5    5  a000002                 count:  DS.W    1
   6    6  a000004                         LONGEVEN
   7    7  a000004                 value:  DS.L    1
   8    8          0000 0008   size:   EQU     *
   9    9
  10   10                              ;; "object" data
  11   11                      data:   SECTION
  12   12   000000             struct: DS.B    size
  13   13
  14   14                              ;; increment data->count
  15   15                      code:   SECTION
  16   16
  17   17   000000 F2xx                LDL     R2, #%XGATE_8(data)
  18   18   000002 FAxx                LDH     R2, #%XGATE_8_H(data)
  19   19   000004 4B42                LDW     R3, (R2, #count)
  20   20   000006 E301                ADDL    R3, #1
  21   21   000008 5B42                STW     R3, (R2, #count)
```

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

ORG - Set location counter

Description

The ORG directive sets the location counter to the value specified by <expression>. Subsequent statements are assigned memory locations starting with the new location counter value. The <expression> must be absolute and may not contain any forward, undefined, or external references. The ORG directive generates an internal section, which is absolute (see Sections).

After the <expression>, LOGICAL is used to make the address of the location counter a logical address.

Syntax

```
ORG <expression> LOGICAL
```

Synonym

None

Example

```
        ORG     $FD2000 LOGICAL
b1:     NOP
b2:     DC.L    b2 + 4
```

Label b1 is located at address $FD2000 and label b2 at address $FD2004:

```
1    1                              ORG     $FD2000 LOGICAL
2     2  aFD2000 0100         b1:     NOP
3     3  aFD2002 00FD 2006    b2:     DC.L    b2 + 4
```

See also

- DC - Define constant
- DCB - Define constant block
- DS - Define space
- SECTION - Declare relocatable section

PAGE - Insert page break

Description

Insert a page break in the assembly listing.

Syntax

```
PAGE
```

Synonym

None

Example

The portion of code in Listing 8.41 demonstrates the use of a page break in the assembler output listing.

Listing 8.41 Example Assembly Source Code

```
code:   SECTION
        DC.B  $00,$12
        DC.B  $00,$34
        PAGE
        DC.B  $00,$56
        DC.B  $00,$78
```

The effect of the PAGE directive can be seen in Listing 8.42.

Listing 8.42 Assembler Output Listing from the Source Code in Listing 8.41

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                      code:   SECTION
    2    2   000000 0012                DC.B  $00,$12
    3    3   000002 0034                DC.B  $00,$34


 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    5    5   000004 0056                DC.B    $00,$56
    6    6   000006 0078                DC.B    $00,$78
```

PLEN - Set page length

Description

Sets the listings page length to `<n>` lines. `<n>` may range from `10` to `10000`. If the number of lines already listed on the current page is greater than or equal to `<n>`, listing will continue on the next page with the new page length setting.

The default page length is 65 lines.

Syntax

```
PLEN<n>
```

Synonym

None

SECTION - Declare relocatable section

Description

This directive declares a relocatable section and initializes the location counter for the following code. The first `SECTION` directive for a section sets the location counter to zero. Subsequent `SECTION` directives for that section restore the location counter to the value that follows the address of the last code in the section.

`<name>` is the name assigned to the section. Two `SECTION` directives with the same name specified refer to the same section.

`<number>` is optional.

A section is a code section when it contains at least one assembly instruction. It is considered to be a constant section if it contains only `DC` or `DCB` directives. A section is considered to be a data section when it contains at least a `DS` directive or if it is empty.

Syntax

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

Synonym

None

Example

The following example illustrates how a section named ‘join’ can be made up of several pieces:

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                      join:   SECTION
   2    2   000000 1234 5678   one:    DC.L    $12345678
   3    3   000004 0100                nop
   4    4
   5    5                      other:  SECTION
   6    6   000000 90AB CDEF   two:    DC.L    $90ABCDEF
   7    7
   8    8                      join:   SECTION
   9    9   000006 F2xx                LDL     R2, #%XGATE_8(one)
  10   10   000008 FAxx                LDH     R2, #%XGATE_8_H(one)
  11   11   00000A 6440                LDB     R4, (R2, R0)
```

See also

- ORG - Set location counter
- DC - Define constant
- DCB - Define constant block
- DS - Define space

SET - Set symbol value

Description

Similar to the EQU - Equate symbol value directive, the `SET` directive assigns the value of the `<expression>` in the operand field to the symbol in the `<label>` field. The `<label>` is an assembly time constant. `SET` does not generate any machine code.

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

Syntax

```
<label>: SET <expression>
```

Synonym

None

Example

See Listing 8.43 for examples of the SET directive.

Listing 8.43 Using the SET Assembler Directive

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1          0000 0002   count:  SET   2
   2    2   000000 02          one:    DC.B  count
   3    3
   4    4          0000 0001   count:  SET   count-1
   5    5   000001 01                  DC.B  count
   6    6
   7    7          0000 0001           IFNE  count
   8    8          0000 0000   count:  SET   count-1
   9    9                              ENDIF
  10   10   000002 00                  DC.B  count
```

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

SPC - Insert blank lines

Description

Inserts `<count>` blank lines in the assembly listing. `<count>` may range from 0 to 65. This has the same effect as writing that number of blank lines in the assembly source. A blank line is a line containing only a carriage return.

Syntax

```
SPC<count>
```

Synonym

None

TABS - Set tab length

Description

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

Syntax

```
TABS <n>
```

Synonym

None

TITLE - Provide listing title

Description

Print the `<title>` on the head of every page of the listing file. This directive must be the first source code line. A title consists of a string of characters enclosed in quotes `(")`.

The title specified will be written on the top of each page in the assembly listing file.

Syntax

```
TITLE "title"
```

Synonym

```
TTL
```

XDEF - External symbol definition

Description

This directive specifies labels defined in the current module that are to be passed to the linker as labels that can be referenced by other modules linked to the current module.

The number of symbols enumerated in an `XDEF` directive is only limited by the memory available at assembly time.

Syntax

```
XDEF [.<size>] <label>[,<label>]...
```

where <size> = W or L (L is the default)

`Synonym`

`GLOBAL, PUBLIC`

Example

See Listing 8.44 for the case where the `XDEF` assembler directive can specify symbols that can be used by other modules.

Listing 8.44 Using XDEF to Create a Variable for Use in Another File

```
        XDEF Count, main
        ;; variable Count can be referenced in other modules,
        ;; same for label main. Note that Linker & Assembler
        ;; are case-sensitive, i.e., Count != count.

Count:  DS.W  2

code:   SECTION
main:   DC.B 1
```

XREF - External symbol reference

Description

This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 32-bit values is passed to the linker.

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

Syntax

```
XREF [.<size>] <symbol>[,<symbol>]...
```

```
where <size> = W or L (default)
```

Synonym

```
EXTERNAL
```

Example

```
XREF OtherGlobal ; Reference "OtherGlobal" defined in
                 ; another module. (See the XDEF
                ; directive example.)
```

9

Macros

A macro is a template for a code sequence. Once a macro is defined, subsequent references to the macro name are replaced by its code sequence.

Macro Overview

A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently called.

The Assembler expands the macro definition each time the macro is called. The macro call causes source statements to be generated, which may include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Calling previously defined macros is also allowed. Source statements generated by a macro call are inserted in the source file at the position where the macro is invoked.

To call a macro, write the macro name in the operation field of a source statement. Place the arguments in the operand field. The macro may contain conditional assembly directives that cause the Assembler to produce in-line-coding variations of the macro definition.

Macros call produces in-line code to perform a predefined function. Each time the macro is called, code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program.

Defining a Macro

The definition of a macro consists of four parts:

- The header statement, a `MACRO` directive with a label that names the macro.
- The body of the macro, a sequential list of assembler statements, some possibly including argument placeholders.
- The `ENDM` directive, terminating the macro definition.
- eventually an instruction `MEXIT`, which stops macro expansion.

See the Assembler Directives chapter for information about the `MACRO`, `ENDM`, `MEXIT`, and `MLIST` directives.

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

Calling Macros

The form of a macro call is:

```
[<label>:] <name>[.<sizearg>] [<argument> [,<argument>]...]
```

Although a macro may be referenced by another macro prior to its definition in the source module, a macro must be defined before its first call. The name of the called macro must appear in the operation field of the source statement. Arguments are supplied in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the call, according to the macro definition and the arguments specified in the macro call. The source statements of the expanded macro are then assembled subject to the same conditions and restrictions affecting any source statement. Nested macros calls are also expanded at this time.

Macro Parameters

As many as 36 different substitutable parameters can be used in the source statements that constitute the body of a macro. These parameters are replaced by the corresponding arguments in a subsequent call to that macro.

A parameter designator consists of a backlashes character (\), followed by a digit (0 - 9) or an uppercase letter (A - Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Consider the macro definition in Listing 9.1:

Listing 9.1 Example Macro Definition

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

When this macro is used in a program, e.g.:

```
    MyMacro.B $10, $56
```

the Assembler expands it to:

```
    DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designator `\1` through `\9` and `\A` through `\Z`, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted for parameter designators in the body of the macro as literal (string) substitutions. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled in line.

It is possible to specify a null argument in a macro call by a comma with no character (not even a space) between the comma and the preceding macro name or comma that follows an argument. When a null argument is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

Macro Argument Grouping

To pass text including commas as a single macro argument, the Assembler supports a special syntax. This grouping starts with the [? prefix and ends with the ?] suffix. If the [? or ?] patterns occur inside of the argument text, they have to be in pairs. Alternatively, brackets, question marks and backward slashes can also be escaped with a backward slash as prefix.

NOTE This escaping only takes place inside of `[? ?]` arguments. A backslash is only removed in this process if it is just before a bracket `([])`, a question mark `(?)`, or a second backslash `(\)`.

Listing 9.2 Example Macro Definition

```
MyMacro:  MACRO
            DC     \1
          ENDM
MyMacro1: MACRO
            \1
          ENDM
```

Listing 9.3 has some macro calls with rather complicated arguments:

Listing 9.3 Macro Calls for Listing 9.2

```
MyMacro [?$10, $56?]
MyMacro [?"\[?"?]
MyMacro1 [?MyMacro  [?$10, $56?]?]
MyMacro1 [?MyMacro \[?$10, $56\?]?]
```

These macro calls expand to the following lines (Listing 9.4):

Listing 9.4 Macro Expansion of Listing 9.3

```
DC    $10, $56
DC    "[?"
DC    $10, $56
DC    $10, $56
```

The Macro Assembler does also supports for compatibility with previous version's macro grouping with an angle bracket syntax (Listing 9.5):

Listing 9.5 Angle Bracket Syntax

```
      MyMacro <$10, $56>
```

However, this old syntax is ambiguous as < and > are also used as compare operators. For example, the following code (Listing 9.6) does not produce the expected result:

Listing 9.6 Potential Problem Using the Angle-Bracket Syntax

```
      MyMacro <1 > 2, 2 > 3> ; Wrong!
```

Because of this the old angle brace syntax should be avoided in new code. There is also and option to disable it explicitly.

See also the -CMacBrackets: Square brackets for macro arguments grouping and the -CMacAngBrack: Angle brackets for grouping macro arguments assembler options.

Labels Inside Macros

To avoid the problem of multiple-defined labels resulting from multiple calls to a macro that has labels in its source statements, the programmer can direct the Assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form _nnnnn where nnnnn is a 5-digit value. The programmer requests an assembler-generated label by specifying `\@` in a label field within a macro body. Each successive label definition that specifies a \@ directive generates a successive value of _nnnnn, thereby creating a unique label on each macro call. Note that \@ may be preceded or followed by additional characters for clarity and to prevent ambiguity.

```
delay:  MACRO
        LDL    R4,#\1
\@LOOP: DEC    R4
        BNE    \@LOOP
        ENDM

        ORG     $FD1000 LOGICAL
entry:
        delay   20
        delay   $40
```

The two macro calls of `delay` are expanded in the following manner:

```
   1    1                        delay:  MACRO
   2    2                                LDL    R4,#\1
   3    3                        \@LOOP: DEC    R4
   4    4                                BNE    \@LOOP
   5    5                                ENDM
   6    6
   7    7                                ORG     $FD1000 LOGICAL
   8    8                        entry:
   9    9                                delay   20
  10    2m aFD1000 F414         +        LDL    R4,#20
  11    3m aFD1002 C401         +_00001LOOP: DEC    R4
  12    4m aFD1004 25FE         +        BNE    _00001LOOP
  13   10                                delay   $40
  14    2m aFD1006 F440         +        LDL    R4,#$40
  15    3m aFD1008 C401         +_00002LOOP: DEC    R4
  16    4m aFD100A 25FE         +        BNE    _00002LOOP
```

Macro Expansion

When the Assembler reads a statement in a source program calling a previously defined macro, it processes the call as described in the following paragraphs.

The symbol table is searched for the macro name. If it is not in the symbol table, an undefined symbol error message is issued.

The rest of the line is scanned for arguments. Any argument in the macro call is saved as a literal or null value in one of the 35 possible parameter fields. When the number of arguments in the call is less than the number of parameters used in the macro the arguments still undefined 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.

Nested Macros

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

A macro call is limited to the length of one line, i.e., 1024 characters.

10

Assembler Listing File

The assembly listing file is the output file of the Assembler that contains information about the generated code. The listing file is generated when the –L assembler option is activated. When an error is detected during assembling from the file, no listing file is generated.

The amount of information available depends upon the following assembler options:

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

The information in the listing file also depends on following assembler directives:

- LIST - Enable listing
- NOLIST - Disable listing
- CLIST - List conditional assembly
- MLIST - List macro expansions

The format from the listing file is influenced by the following assembler directives:

- PLEN - Set page length
- LLEN - Set line length
- TABS - Set tab length
- SPC - Insert blank lines
- PAGE - Insert page break
- NOPAGE - Disable paging
- TITLE - Provide listing title.

The name of the generated listing file is `<base name>.lst.`

Page Header

The page header consists of three lines:

- The first line contains an optional user string defined in the `TITLE` directive.
- The second line contains the name of the Assembler vendor (`Freescale`) as well as the target processor name - HCS08.
- The third line contains a copyright notice.

Listing 10.1 Example Page Header Output

```
Demo Application
Freescale HCS08-Assembler
(c) COPYRIGHT Freescale 1991-2009
```

Source Listing

The printed columns can be configured in various formats with the -Lasmc: Configure listing file assembler option. The default format of the source listing contains the five columns: Abs. Rel. Loc Obj.code and Source line.

Abs.

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

Listing 10.2 Example Output Listing - Abs. Column

```
 Abs. Rel.   Loc    Obj. code   Source line
 ---- ----   ------ ---------   -----------
    1    1                                XDEF entry
    2    2                                INCLUDE "opt_l.inc"
    3    1i                    ; getadr16 Dest Register, Variable Name
    4    2i                     getadr16: MACRO
    5    3i                               LDL \1, #%XGATE_8(\2)
    6    4i                               LDH \1, #%XGATE_8_H(\2)
    7    5i                               ENDM
    8    3
    9    4                      myData:   SECTION
   10    5   000000             a:        DS.B  1
   11    6
```

```
  12    7                      myConst:  SECTION
  13    8   000000 1234        init:     DC.W  $1234
  14    9
  15   10                      myCode:   SECTION
  16   11                      entry:
  17   12                                getadr16 R2, init
  18    3m  000000 F2xx       +          LDL R2, #%XGATE_8(init)
  19    4m  000002 FAxx       +          LDH R2, #%XGATE_8_H(init)
  20   13   000004 6440                  LDB      R4, (R2, R0)
  21   14
  22   15   000006 E401                  ADDL     R4, #1
  23   16
  24   17                                getadr16 R2, a
  25    3m  000008 F2xx       +          LDL R2, #%XGATE_8(a)
  26    4m  00000A FAxx       +          LDH R2, #%XGATE_8_H(a)
  27   18   00000C 7440                  STB      R4, (R2, R0)
  28   19
  29   20                      loop:
  30   21   00000E 3FFF                  BRA loop
```

Rel.

This column contains the relative line number for each instruction. The relative line number is the line number in the source file. For included files, the relative line number is the line number in the included file. For macro call expansion, the relative line number is the line number of the instruction in the macro definition. See Listing 10.3.

An `'i'` suffix is appended to the relative line number when the line comes from an included file. An `'m'` suffix is appended to the relative line number when the line is generated by a macro call.

Listing 10.3 Example Listing File - Rel. Column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                                XDEF entry
   2    2                                INCLUDE "opt_l.inc"
   3    1i                    ; getadr16 Dest Register, Variable Name
   4    2i                     getadr16: MACRO
   5    3i                               LDL \1, #%XGATE_8(\2)
   6    4i                               LDH \1, #%XGATE_8_H(\2)
   7    5i                               ENDM
   8    3
   9    4                      myData:   SECTION
  10    5   000000             a:        DS.B  1
  11    6
  12    7                      myConst:  SECTION
  13    8   000000 1234        init:     DC.W  $1234
  14    9
  15   10                      myCode:   SECTION
  16   11                      entry:
  17   12                                getadr16 R2, init
  18    3m  000000 F2xx       +          LDL R2, #%XGATE_8(init)
  19    4m  000002 FAxx       +          LDH R2, #%XGATE_8_H(init)
  20   13   000004 6440                  LDB      R4, (R2, R0)
  21   14
  22   15   000006 E401                  ADDL     R4, #1
  23   16
  24   17                                getadr16 R2, a
  25    3m  000008 F2xx       +          LDL R2, #%XGATE_8(a)
  26    4m  00000A FAxx       +          LDH R2, #%XGATE_8_H(a)
  27   18   00000C 7440                  STB      R4, (R2, R0)
  28   19
  29   20                      loop:
  30   21   00000E 3FFF                  BRA loop
```

In the previous example, the line number displayed in the ‘Rel.’ column represents the line number of the corresponding instruction in the source file.

Loc

This column contains the address of the instruction. For absolute sections, the address is preceded by an ‘a’ and contains the absolute address of the instruction. For relocatable sections, this address is the offset of the instruction from the beginning of the relocatable section. This offset is a hexadecimal number coded on 6 digits.

A value is written in this column in front of each instruction generating code or allocating storage. This column is empty in front of each instruction that does not generate code (for example `SECTION`, `XDEF`, …). See Listing 10.4.

Listing 10.4 Example Listing File - Loc Column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                                XDEF entry
   2    2                                INCLUDE "opt_l.inc"
   3    1i                    ; getadr16 Dest Register, Variable Name
   4    2i                     getadr16: MACRO
   5    3i                               LDL \1, #%XGATE_8(\2)
   6    4i                               LDH \1, #%XGATE_8_H(\2)
   7    5i                               ENDM
   8    3
   9    4                          myData:   SECTION
  10    5   000000                 a:        DS.B  1
  11    6
  12    7                          myConst:  SECTION
  13    8   000000 1234            init:     DC.W  $1234
  14    9
  15   10                          myCode:   SECTION
  16   11                          entry:
  17   12                                    getadr16 R2, init
  18    3m  000000 F2xx        +             LDL R2, #%XGATE_8(init)
  19    4m  000002 FAxx        +             LDH R2, #%XGATE_8_H(init)
  20   13   000004 6440                      LDB      R4, (R2, R0)
  21   14
  22   15   000006 E401                      ADDL     R4, #1
  23   16
  24   17                                    getadr16 R2, a
  25    3m  000008 F2xx        +             LDL R2, #%XGATE_8(a)
  26    4m  00000A FAxx        +             LDH R2, #%XGATE_8_H(a)
  27   18   00000C 7440                      STB      R4, (R2, R0)
  28   19
  29   20                          loop:
  30   21   00000E 3FFF                      BRA loop
```

There is no location counter specified in front of the instruction `INCLUDE opt_l.inc` because this instruction does not generate code.

Obj. Code

This column contains the hexadecimal code of each instruction in hexadecimal format. This code is not identical to the code stored in the object file. The letter ‘x’ is displayed at the position where the address of an external or relocatable label is expected. Code at any position when ‘x’ is written will be determined at link time. See Listing 10.5.

Listing 10.5 Example Listing File - Obj. code Column

```
Abs. Rel.   Loc    Obj. code   Source line
---- ----   ------ ---------   -----------
   1    1                                XDEF entry
   2    2                                INCLUDE "opt_l.inc"
   3    1i                    ; getadr16 Dest Register, Variable Name
   4    2i                     getadr16: MACRO
   5    3i                               LDL \1, #%XGATE_8(\2)
   6    4i                               LDH \1, #%XGATE_8_H(\2)
   7    5i                               ENDM
   8    3
```

```
    9    4                         myData:   SECTION
   10    5   000000                a:        DS.B   1
   11    6
   12    7                         myConst:  SECTION
   13    8   000000 1234           init:     DC.W   $1234
   14    9
   15   10                         myCode:   SECTION
   16   11                         entry:
   17   12                                   getadr16 R2, init
   18    3m  000000 F2xx         +           LDL R2, #%XGATE_8(init)
   19    4m  000002 FAxx         +           LDH R2, #%XGATE_8_H(init)
   20   13   000004 6440                     LDB      R4, (R2, R0)
   21   14
   22   15   000006 E401                     ADDL     R4, #1
   23   16
   24   17                                   getadr16 R2, a
   25    3m  000008 F2xx         +           LDL R2, #%XGATE_8(a)
   26    4m  00000A FAxx         +           LDH R2, #%XGATE_8_H(a)
   27   18   00000C 7440                     STB      R4, (R2, R0)
   28   19
   29   20                          loop:
   30   21   00000E 3FFF                     BRA loop
```

Source Line

This column contains the source statement. This is a copy of the source line from the source module. For lines resulting from a macro expansion, the source line is the expanded line, where parameter substitution has been done. See Listing 10.6.

Listing 10.6 Example Listing File - Source line Column

```
  Abs. Rel.   Loc    Obj. code   Source line
  ---- ----   ------ ---------   -----------
    1    1                                 XDEF entry
    2    2                                 INCLUDE "opt_1.inc"
    3    1i                      ; getadr16 Dest Register, Variable Name
    4    2i                       getadr16: MACRO
    5    3i                                 LDL \1, #%XGATE_8(\2)
    6    4i                                 LDH \1, #%XGATE_8_H(\2)
    7    5i                                 ENDM
    8    3
    9    4                        myData:   SECTION
   10    5   000000               a:        DS.B   1
   11    6
   12    7                        myConst:  SECTION
   13    8   000000 1234         init:     DC.W  $1234
   14    9
   15   10                       myCode:   SECTION
   16   11                       entry:
   17   12                                 getadr16 R2, init
   18    3m  000000 F2xx       +           LDL R2, #%XGATE_8(init)
   19    4m  000002 FAxx       +           LDH R2, #%XGATE_8_H(init)
   20   13   000004 6440                   LDB      R4, (R2, R0)
   21   14
   22   15   000006 E401                   ADDL     R4, #1
   23   16
   24   17                                 getadr16 R2, a
   25    3m  000008 F2xx       +           LDL R2, #%XGATE_8(a)
   26    4m  00000A FAxx       +           LDH R2, #%XGATE_8_H(a)
   27   18   00000C 7440                   STB      R4, (R2, R0)
   28   19
   29   20                       loop:
   30   21   00000E 3FFF                   BRA loop
```

11

Mixed C and Assembler Applications

When you intend to mix Assembly source file and ANSI-C source files in a single application, the following issues are important:

- Memory Models
- Parameter Passing Scheme
- Return Value
- Accessing Assembly Variables in an ANSI-C Source File
- Accessing ANSI-C Variables in an Assembly Source File
- Invoking an Assembly Function in an ANSI-C Source File
- Support for Structured Types

To build mixed C and Assembler applications, you have to know how the C Compiler uses registers and calls procedures. The following sections will describe this for compatibility with the compiler. If you are working with another vendor's ANSI-C compiler, refer to your Compiler Manual to get the information about parameter passing rules.

Memory Models

The XGATE assembler does not support different memory models.

Parameter Passing Scheme

Please check the compiler manual, back-end chapter about the details of parameter passing.

Return Value

Please check the compiler manual's backend chapter about the details of parameter passing.

Accessing Assembly Variables in an ANSI-C Source File

A variable or constant defined in an assembly source file is accessible in an ANSI-C source file.

The variable or constant is defined in the assembly source file using the standard assembly syntax.

Variables and constants must be exported using the `XDEF` directive to make them visible from other modules (Listing 11.1).

Listing 11.1 Example of Data and Constant Definition

```
          XDEF  ASMData, ASMConst
DataSec:  SECTION
ASMData:  DS.W  1      ; Definition of a variable
ConstSec: SECTION
ASMConst: DC.W  $44A6  ; Definition of a constant
```

We recommend that you generate a header file for each assembler source file. This header file should contain the interface to the assembly module.

An external declaration for the variable or constant must be inserted in the header file (Listing 11.2).

Listing 11.2 Example of Data and Constant Declarations

```
/* External declaration of a variable */
extern int       ASMData;
/* External declaration of a constant */
extern const int ASMConst;
```

The variables or constants can then be accessed in the usual way, using their names (Listing 11.3).

Listing 11.3 Example of Data and Constant Reference

```
  ASMData = ASMConst + 3;
```

Accessing ANSI-C Variables in an Assembly Source File

A variable or constant defined in an ANSI-C source file is accessible in an assembly source file.

The variable or constant is defined in the ANSI-C source file using the standard ANSI-C syntax (Listing 11.4).

Listing 11.4 Example Definition of Data and Constants

```
unsigned int CData;        /* Definition of a variable */
unsigned const int CConst; /* Definition of a constant */
```

An external declaration for the variable or constant must be inserted into the assembly source file (Listing 11.5).

This can also be done in a separate file, included in the assembly source file.

Listing 11.5 Example declaration of data and constants

```
XREF CData;  External declaration of a variable
XREF CConst; External declaration of a constant
```

The variables or constants can then be accessed in the usual way, using their names (Listing 11.6).

NOTE The compiler supports the automatic generation of assembler include files. See the description of the `-La` compiler option in the compiler manual.

Listing 11.6 Example of Data and Constant Reference

```
        LDL R2, #%XGATE_8(CConst)
        LDH R2, #%XGATE_8_H(CConst)
        LDB R4, (R2, R0)
         ....
        LDL R2, #%XGATE_8(CData)
        LDH R2, #%XGATE_8_H(CData)
        LDB R4, (R2, R0)
       ....
```

Invoking an Assembly Function in an ANSI-C Source File

A function implemented in an assembly source file can be invoked in a C source file (Listing 11.7). During the implementation of the function in the assembly source file, you should pay attention to the parameter passing scheme of the ANSI-C compiler you are using in order to retrieve the parameter from the right place.

When mixing C and assembly code, the XGATE compiler name mangling scheme adds a __X_ prefix for each function. Therefore, to define a function in assembly that will be called from C, the exported label has to have a __X_ prefix that is implicitly added on the C side.

Example of Assembly File

```
CodeSec: SECTION
__X_AddVar:
        LDL    R3, #%XGATE_8(CData)
        LDH    R3, #%XGATE_8_H(CData)
        LDB    R4, (R3, R0)
        ADD    R2, R2, R4

        LDL    R3, #%XGATE_8(ASMData)
        LDH    R3, #%XGATE_8_H(ASMData)
        STB    R2, (R3, R0)

        JAL    R6

(C)
extern void AddVar(int val);
...
```

Example of a C File

A C source code file (`mixc.c`) has the `main()` function which calls the `AddVar()` function. See Listing 11.7. (Compile it with the `-Cc` compiler option when using the HIWARE Object File Format).

Listing 11.7 Example C Source Code File: mixc.c

```
#include "mixasm.h"

const unsigned char CData=12;
```

```
volatile int Error;

void main (void) {
  AddVar (10);
  if (ASMData != CData + 10) {
    Error = 1;
  } else {
    Error = 0;
  }
  for (;;) { /* always */ }
}
```

CAUTION The Assembler will not check the number and type of function parameters.

We recommend that you generate a header file for each assembly source file. This header file (`mixasm.h` in Listing 11.8) should contain the interface to the assembly module.

Listing 11.8 Header File for the Assembly File: mixasm.h

```
/* mixasm.h */
#ifndef _MIXASM_H_
#define _MIXASM_H_

void AddVar(unsigned char value);
/* function that adds the parameter value to global CData */
/* and then stores the result in ASMData */

/* variable which receives the result of AddVar */
extern char ASMData;

#endif /* _MIXASM_H_ */
```

The function can then be invoked in the usual way, using its name.

The application must be correctly linked.

For these `C` and `*.asm` files, a possible linker parameter file is shown in Listing 11.9.

Listing 11.9 Example of Linker Parameter File: mixasm.prm

```
LINK mixasm.abs
NAMES
  mixc.o mixasm.o
END
SECTIONS
```

```
  MY_ROM   = READ_ONLY  0x4000 TO 0x4FFF ALIGN 2;
  MY_RAM   = READ_WRITE 0x2400 TO 0x2FFF ALIGN 2;
  MY_STACK = READ_WRITE 0x2000 TO 0x23FF;
END
PLACEMENT
  DEFAULT_RAM    INTO MY_RAM;
  DEFAULT_ROM    INTO MY_ROM;
  SSTACK         INTO MY_STACK;
END
INIT main
```

NOTE We recommend that you use the same memory model and object file format for all the generated object files.

Support for Structured Types

When the -Struct: Support for structured types assembler option is activated, the Macro Assembler also supports the definition and usage of structured types. This allows an easier way to access ANSI-C structured variables in the Macro Assembler.

In order to provide an efficient support for structured type the macro assembler should provide notation to:

- Define a structured type. See Structured Type Definition.
- Define a structured variable. See Variable Definition.
- Declare a structured variable. See Variable Declaration.
- Access the address of a field inside of a structured variable. See Accessing a Field Address
- Access the offset of a field inside of a structured variable. See Accessing a field offset.

NOTE Some limitations apply in the usage of the structured types in the Macro Assembler. See Structured Type: Limitations.

Structured Type Definition

The Macro Assembler is extended with the following new keywords in order to support ANSI-C type definitions.

- `STRUCT`
- `UNION`

The structured type definition for `STRUCT` can be encoded as in Listing 11.10:

Listing 11.10 Definition for STRUCT

```
typeName: STRUCT
   lab1: DS.W  1   lab2: DS.W  1         ...
   ENDSTRUCT
```

where:

`'typeName'` is the name associated with the defined type. The type name is considered to be a user-defined keyword. The Macro Assembler will be case-insensitive on `typeName`.

`'STRUCT'` specifies that the type is a structured type.

`'lab1'`and `'lab2'` are the fields defined inside of the `'typeName' type`. The fields will be considered as user-defined labels, and the Macro Assembler will be case-sensitive on label names.

As with all other directives in the Assembler, the `STRUCT` and `UNION` directives are case-insensitive.

The `STRUCT` and `UNION` directives cannot start on column 1 and must be preceded by a label.

Types Allowed for Structured Type Fields

The field inside of a structured type may be:

- Another structured type or
- A base type, which can be mapped on 1, 2, or 4 bytes.

Table 11.1 shows how the ANSI-C standard types are converted in the assembler notation:

Table 11.1 Converting ANSI-C Standard Types to Assembler Notation

ANSI-C type	Assembler Notation
char	DS - Define space
short	DS.W
int	DS.W
long	DS.L
enum	DS.W
bitfield	-- not supported --
float	DS.F
double	DS.D
data pointer	DS.W or DS.L
function pointer	-- not supported --

Variable Definition

The Macro Assembler can provide a way to define a variable with a specific type. This is done using `var: typeName` where `var` is the name of the variable and `typeName` is the type associated with the variable. See Listing 11.11.

Listing 11.11 Assembly Code Analog of a C struct of Type: myType

```
myType:    STRUCT
field1:    DS.W 1
field2:    DS.W 1
field3:    DS.B 1
field4:    DS.B 3
field5:    DS.W 1
           ENDSTRUCT

DataSection: SECTION
structVar:   TYPE myType ; var ‘structVar’ is of type ‘myType’
```

Variable Declaration

The Macro Assembler can provide a way to associated a type with a symbol which is defined externally. This is done by extending the `XREF` syntax:

```
XREF var: typeName, var2
```

where:

- `var` is the name of an externally defined symbol.
- `typeName` is the type associated with the variable `var`.

`var2` is the name of another externally defined symbol. This symbol is not associated with any type. See Listing 11.12 for an example.

Listing 11.12 Example of Extending XREF

```
myType: STRUCT
field1:   DS.W 1
field2:   DS.W 1
field3:   DS.B 1
field4:   DS.B 3
field5:   DS.W 1
        ENDSTRUCT

        XREF extData: myType ; var ‘extData’ is type ‘myType’
```

Accessing a Structured Variable

The Macro Assembler can provide a means to access each structured type field absolute address and offset.

Accessing a Field Address

To access a structured-type field address (Listing 11.13), the Assembler uses the colon character ':'.

`var:field`

where

- '`var`' is the name of a variable, which was associated with a structured type.
- '`field`' is the name of a field in the structured type associated with the variable.

Listing 11.13 Example of Accessing a Field Address

```
myType:   STRUCT
field1:     DS.W  1
field2:     DS.W  1
field3:     DS.B  1
field4:     DS.B  3
field5:     DS.W  1
          ENDSTRUCT

          XREF  myData:myType
          XDEF  entry

CodeSec:  SECTION
entry:
          LDL     R2, #%XGATE_8(myData:field3)
          LDH     R2, #%XGATE_8_H(myData:field3)
          LDB     R4, (R2, R0)
          ;; loads byte myData.field3
```

NOTE The period cannot be used as a separator because in assembly language it is a valid character inside a symbol name.

Accessing a field offset

To access a structured type field offset, the Assembler will use the following notation:

`<typeName>-><field>`

where:

- `typeName` is the name of a structured type.
- `field` is the name of a field in the structured type associated with the variable. See Listing 11.14 for an example of using this notation for accessing an offset.

Listing 11.14 Accessing a Field Offset with the -><field> Notation

```
myType:   STRUCT
field1:     DS.W  1
field2:     DS.W  1
field3:     DS.B  1
field4:     DS.B  3
field5:     DS.W  1
          ENDSTRUCT
          XREF.B myData
```

```
         XDEF   entry

CodeSec: SECTION
entry:
         LDL     R2, #%XGATE_8(myType->field3)
         LDH     R2, #%XGATE_8_H(myType->field3)
         LDB     R4, (R2, R0)
         ;; loads byte myData.field3
```

Structured Type: Limitations

A field inside of a structured type may be:

- Another structured type
- A base type, which can be mapped on 1, 2, or 4 bytes.

The Macro Assembler is not able to process bitfields or pointer types.

The type referenced in a variable definition or declaration must be defined previously. A variable cannot be associated with a type defined afterwards.

12

Make Applications

This chapters has the following sections:

- Assembly Applications
- Memory Maps and Segmentation

Assembly Applications

This section covers:

- Directly Generating an Absolute File
- Mixed C and Assembly Applications

Directly Generating an Absolute File

When an absolute file is directly generated by the Assembler:

- The application entry point must be specified in the assembly source file using the directive `ABSENTRY`.
- The whole application must be encoded in a single assembly unit.
- The application should only contain absolute sections.

Generating Object Files

The entry point of the application must be mentioned in the Linker parameter file using the "`INIT` *funcname*" command. The application is build of the different object files with the Linker. The Linker is document in a separate document.

Your assembly source files must be separately assembled. Then the list of all the object files building the application must be enumerated in the application PRM file.

Mixed C and Assembly Applications

Normally the application starts with the main procedure of a C file. All necessary object files - assembly or C - are linked with the Linker in the same fashion like pure C applications. The Linker is documented in a separate document.

Memory Maps and Segmentation

Relocatable Code Sections are placed in the `DEFAULT_ROM` or `.text` Segment.

Relocatable Data Sections are placed in the `DEFAULT_RAM` or `.data` Segment.

NOTE The `.text` and `.data` names are only supported when the ELF object file format is used.

There are no checks at all that variables are in `RAM`. If you mix code and data in a section you cannot place the section into `ROM`. That is why we suggest that you separate code and data into different sections.

If you want to place a section in a specific address range, you have to put the section name in the placement portion of the linker parameter file (Listing 12.1).

Listing 12.1 Example Assembly Source Code

```
SECTIONS
  ROM1       = READ_ONLY  0x0200 TO 0x0FFF ALIGN 2;
  SpecialROM = READ_ONLY  0x8000 TO 0x8FFF ALIGN 2;
  RAM        = READ_WRITE 0x4000 TO 0x4FFF ALIGN 2;
PLACEMENT
  DEFAULT_ROM   INTO ROM1;
  mySection     INTO SpecialROM;
  DEFAULT_RAM   INTO RAM;
END
```

13

Working with Sections, Vectors, and Modules

This chapter covers the following topics:

- Working with Absolute Sections
- Work with Relocatable Sections
- Initializing the Vector Table
- Splitting an Application into Different Modules

Working with Absolute Sections

An absolute section is a section whose start address is known at assembly time.

(See modules fiboorg.asm and fiboorg.prm in the demo directory)

Defining Absolute Sections in an Assembly Source File

An absolute section is defined using the ORG directive. In that case, the Macro Assembler generates a pseudo section, whose name is "ORG_<index>", where index is an integer which is incremented each time an absolute section is encountered (Listing 13.1).

Listing 13.1 Defining an Absolute Section Containing Data

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

In the previous portion of code, the label cst1 is located at address $A00, and label cst2 is located at address $A01.

Listing 13.2 Assembler Output Listing for Listing 13.1

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

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used.
- Publish the application's entry point using `XDEF`.
- The programmer should ensure that the addresses specified in the source files are valid addresses for the MCU being used.

Linking an Application Containing Absolute Sections

When the Assembler is generating an object file, applications containing only absolute sections must be linked. The linker parameter file must contain at least:

- The name of the absolute file
- The name of the object file which should be linked
- The specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- The specification of a memory area where the sections containing code or constants must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- The specification of the application entry point, and
- The definition of the reset vector.

The minimal linker parameter file will look as shown in Listing 13.3.

Listing 13.3 Minimal Linker Parameter File

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

```
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file.
*/
  MY_ROM  = READ_ONLY  0x4000 TO 0x4FFF ALIGN 2;
/* 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 ALIGN 2;
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.            */
```

NOTE There should be no overlap between the absolute sections defined in the assembly source file and the memory areas defined in the PRM file.

NOTE As the memory areas (segments) specified in the PRM file are only used to allocate relocatable sections, nothing will be allocated there when the application contains only absolute sections. In that case you can even specify invalid address ranges in the PRM file.

Work with Relocatable Sections

A relocatable section is a section whose start address is determined at linking time.

Defining Relocatable Sections in a Source File

A relocatable section is defined using the `SECTION` directive. See Listing 13.4 for an example of defining relocatable sections.

Listing 13.4 Defining Relocatable Sections Containing Data

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

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

In the previous portion of code, the label `cst1` will be located at an offset 0 from the section `constSec` start address, and label `cst2` will be located at an offset 1 from the section `constSec` start address. See Listing 13.5.

Listing 13.5 Assembler Output Listing for Listing 13.4

```
  2    2                  constSec: SECTION ; Relocatable
  3    3   000000 A6      cst1:     DC.B    $A6
  4    4   000001 BC      cst2:     DC.B    $BC
  5    5
  6    6                  dataSec:  SECTION ; Relocatable
  7    7   000000         var:      DS.B   1
```

In order to avoid problems during linking or execution from an application, an assembly file should at least:

- Initialize the stack pointer if the stack is used
- Publish the application's entry point using the `XDEF` directive.

Linking an Application Containing Relocatable Sections

Applications containing relocatable sections must be linked. The linker parameter file must contain at least:

- The name of the absolute file,
- The name of the object file which should be linked,
- The specification of a memory area where the sections containing variables must be allocated,
- The specification of a memory area where the sections containing code or constants must be allocated,
- The specification of the application's entry point, and
- The definition of the reset vector.

A minimal linker parameter file will look as shown in Listing 13.6.

Listing 13.6 Minimal Linker Parameter File

```
/* Name of the executable file generated.      */
LINK test.abs
/* Name of the object file in the application. */
NAMES
  test.o
END
SECTIONS
/* READ_ONLY memory area.   */
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF ALIGN 2;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF ALIGN 2;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM.          */
  DEFAULT_RAM              INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM. */
  DEFAULT_ROM, constSec    INTO MY_ROM;
END
INIT entry                  /* Application entry point.            */
```

NOTE Ensure that the memory ranges you specify in the `SECTIONS` block are valid addresses for the controller you are using. In addition, when using the SDI debugger the addresses specified for code or constant sections must be located in the target board ROM area. Otherwise, the debugger will not be able to load the application.

Initializing the Vector Table

The vector table can be initialized in the assembly source file.

Listing 13.7 Vector Table Initialization Example

```
  XDEF XGATE_VectorTable    ; for S12X

XGATE_CODE: SECTION

; SCI0Handler variables
  ALIGN 2
DataSCI0:
counter DC.W  0

; interrupt handler for the channel 0x6B (SCI0)
  ALIGN 2
SCI0Handler:
    LDW   R2,(R1,#(counter - DataSCI0))   ; load counter
    INC   R2
    STW   R2,(R1,#(counter - DataSCI0))   ; store counter
    RTS

; interrupt handler for all others
ErrorHandler:
    ; The channel number is in R1
    BRK
    RTS


XGATE_VectorTable:
; Channel_# = Vector address / 2
Channel_00: DC.W %XGATE_16(ErrorHandler), %XGATE_16(  0)      ; Res.
Channel_01: DC.W %XGATE_16(ErrorHandler), %XGATE_16(  1)      ; Res.
...
Channel_6B: DC.W %XGATE_16(SCI0Handler),  %XGATE_16(DataSCI0) ; SCI0
...
Channel_7F: DC.W %XGATE_16(ErrorHandler), %XGATE_16(127)      ; Res.
```

You must set up the **XGATE_VectorTable** variable in the S12(X).

Splitting an Application into Different Modules

Complex applications or applications involving several programmers can be split into several simple modules. In order to avoid any problem when merging the different modules, the following rules must be followed.

For each assembly source file, one include file must be created containing the definition of the symbols exported from this module. For the symbols referring to a code label, a small description of the interface is required.

Example of an Assembly File (Test1.asm)

See Listing 13.8 for an example `Test1.asm` include file.

Listing 13.8 Separating Code into Modules — Test1.asm

```
        XDEF AddSource
        XDEF Source

DataSec: SECTION
Source: DS.B    1

CodeSec: SECTION
AddSource:
        LDL     R3, #%XGATE_8(Source)
        LDH     R3, #%XGATE_8_H(Source)
        LDB     R4, (R3, R0)

        ADD     R2, R2, R4

        STB     R2, (R3, R0)

        JAL     R6
```

Corresponding Include File (Test1.inc)

See Listing 13.9 for an example `Test1.inc` include file.

Listing 13.9 Separating Code into Modules — Test1.inc

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

         XREF Source
; The Source variable is a 1-byte variable.
```

Example of an Assembly File (Test2.asm)

Listing 13.10 is another assembly code file module for this project.

Listing 13.10 Separating Code into Modules—Test2.asm

```
        XDEF    entry
        INCLUDE "Test1.inc"

ConstSec: SECTION
operand1: DC.B  $29
operand2: DC.B  $24

CodeSec: SECTION
entry:
        LDL     R3, #%XGATE_8(operand1)
        LDH     R3, #%XGATE_8_H(operand1)
        LDB     R4, (R3, R0)

        LDL     R3, #%XGATE_8(Source)
        LDH     R3, #%XGATE_8_H(Source)
        STB     R4, (R3, R0)

        LDL     R3, #%XGATE_8(operand2)
        LDH     R3, #%XGATE_8_H(operand2)
        LDB     R2, (R3, R0)
        LDL     R6, #%XGATE_8(AddSource)
        LDH     R6, #%XGATE_8_H(AddSource)
        JAL     R6

        BRA     entry
```

The application's `*.prm` file should list both object files building the application. When a section is present in the different object files, the object file sections are concatenated into a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the `*.prm` file.

Example of a PRM file (Test2.prm)

Listing 13.11 Separating Assembly Code into Modules—Test2.prm

```
LINK test2.abs /* Name of the executable file generated. */
NAMES
  test1.o
  test2.o / *Name of the object files building the application. */
END

SECTIONS
  MY_ROM  = READ_ONLY  0x2B00 TO 0x2BFF ALIGN 2; /* READ_ONLY  mem. */
  MY_RAM  = READ_WRITE 0x2800 TO 0x28FF ALIGN 2; /* READ_WRITE mem. */
END

PLACEMENT
  /* variables are allocated in MY_RAM          */
  DataSec, DEFAULT_RAM           INTO MY_RAM;

  /* code and constants are allocated in MY_ROM */
  CodeSec, ConstSec, DEFAULT_ROM INTO MY_ROM;
END
INIT  entry      /* Definition of the application entry point. */
```

NOTE The `CodeSec` section is defined in both object files. In `test1.o`, the `CodeSec` section contains the symbol `AddSource`. In `test2.o`, the `CodeSec` section contains the `entry` symbol. According to the order in which the object files are listed in the `NAMES` block, the function `AddSource` is allocated first and the `entry` symbol is allocated next to it.

II

Appendices

This document has the following appendices:

- Global Configuration File Entries
- Local Configuration File Entries

A

Global Configuration File Entries

This appendix documents the sections and entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [Installation] Section
- [Options] Section
- [XXX_Assembler] Section
- [Editor] Section

[Installation] Section

Path

Description

Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

Arguments

Last installation path.

Example

```
Path=C:\install
```

Group

Description

Whenever a tool is installed, the installation script stores the installation program group created into this variable.

Arguments

Last installation program group.

Example

```
Group=Assembler
```

[Options] Section

DefaultDir

Description

Specifies the current directory for all tools on a global level. See also DEFAULTDIR: Default current directory environment variable.

Arguments

Default directory to be used.

Example

```
DefaultDir=C:\install\project
```

[XXX_Assembler] Section

This section documents the entries that can appear in an `[`*`XXX`*`_Assembler]` section of the `mcutools.ini` file.

NOTE *XXX* is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC12 Assembler, the name of this section would be [HC12_Assembler].

SaveOnExit

Description

1 if the configuration should be stored when the Assembler is closed, 0 if it should not be stored. The Assembler does not ask to store a configuration in either cases.

Arguments

1/0

SaveAppearance

Description

1 if the visible topics should be stored when writing a project file, 0 if not. The command line, its history, the windows position and other topics belong to this entry.

This entry corresponds to the state of the *Appearance* check box in the Save Configuration dialog box.

Arguments

1/0

SaveEditor

Description

If the editor settings should be stored when writing a project file, 0 if not. The editor setting contain all information of the *Editor Configuration* dialog box.

This entry corresponds to the state of the check box *Editor Configuration* in the Save Configuration Dialog Box.

Arguments

1/0

SaveOptions

Description

1 if the options should be contained when writing a project file, 0 if not.

This entry corresponds to the state of the *Options* check box in the Save Configuration Dialog Box.

Arguments

1/0

RecentProject0, RecentProject1

Description

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Arguments

Names of the last and prior project files

Example

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

[Editor] Section

Editor_Name

Description

Specifies the name of the editor used as global editor. This entry has only a descriptive effect. Its content is not used to start the editor.

Arguments

The name of the global editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Editor_Exe

Description

Specifies the filename which is started to edit a text file, when the global editor setting is active.

Arguments

The name of the executable file of the global editor (including path).

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Editor_Opts

Description

Specifies options (arguments), which should be used when starting the global editor. If this entry is not present or empty, `"%f"` is used. The command line to launch the editor is built by taking the `Editor_Exe` content, then appending a space followed by the content of this entry.

Arguments

The options to use with the global editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration Save Configuration* dialog box.

Example

```
[Editor]

editor_name=IDF

editor_exe=C:\Freescale\prog\idf.exe

editor_opts=%f -g%l,%c
```

Example

Listing A.1 shows a typical `mcutools.ini` file.

Listing A.1 Typical mcutools.ini File Layout

```
[Installation]
Path=c:\Freescale
Group=Assembler

[Editor]
editor_name=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%l,%c

[Options]
DefaultDir=c:\myprj

[HC12_Assembler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```

B

Local Configuration File Entries

This appendix documents the sections and entries that can appear in the local configuration file. Usually, you name this file `project.ini`, where *`project`* is a placeholder for the name of your project.

A *`project`*`.ini` file can contains these sections:

- [Editor] Section
- [XXX_Assembler] Section
- Example

[Editor] Section

Editor_Name

Description

Specifies the name of the editor used as local editor. This entry has only a description effect. Its content is not used to start the editor.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

Arguments

The name of the local editor

Saved

Only with `'Editor Configuration'` set in the *File > Configuration > Save Configuration* dialog box.

Editor_Exe

Description

Specifies the filename with is started to edit a text file, when the local editor setting is active. In the editor configuration dialog box, the local editor selection is only active when this entry is present and not empty.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

Arguments

The name of the executable file of the local editor (including path).

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

Editor_Opts

Description

Specifies options (arguments), which should be used when starting the local editor. If this entry is not present or empty, `"%f"` is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by the content of this entry.

This entry has the same format as for the global editor configuration in the `mcutools.ini` file.

Arguments

The options to use with the local editor

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

Example

```
[Editor]
editor_name=IDF
editor_exe=C:\Freescale\prog\idf.exe
editor_opts=%f -g%l,%c
```

[XXX_Assembler] Section

This section documents the entries that can appear in an `[`*`XXX`*`_Assembler]` section of a `project.ini` file.

NOTE *XXX* is a placeholder for the name of the name of the particular Assembler you are using. For example, if you are using the HC12 Assembler, the name of this section would be `[HC12_Assembler]`.

RecentCommandLineX, X= integer

Description

This list of entries contains the content of the command line history.

Arguments

String with a command line history entry, e.g., `fibo.asm`

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

CurrentCommandLine

Description

The currently visible command line content.

Arguments

String with the command line, e.g., "`fibo.asm -w1`"

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

StatusbarEnabled

Description

Current statusbar state.

- 1: Statusbar is visible
- 0: Statusbar is hidden

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it any more.

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

ToolbarEnabled

Description

Current toolbar state:

- 1: Toolbar is visible
- 0: Toolbar is hidden

Arguments

1/0

Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

WindowPos

Description

This numbers contain the position and the state of the window (maximized, etc.) and other flags.

Arguments

10 integers, e.g., "`0,1,-1,-1,-1,-1,390,107,1103,643`"

Special

This entry is only considered at startup. Afterwards, any load operations do not use it any longer.

Changes of this entry do not show the `"*"` in the title.

Saved

Only with *Appearance* set in the *File* > *Configuration* > *Save Configuration* dialog box.

WindowFont

Description

Font attributes.

Arguments

`size:` = 0 -> generic size, < 0 -> font character height, > 0 -> font cell height

`weight`: 400 = normal, 700 = bold (valid values are 0..1000)

`italic`: 0 = no, 1 = yes

`font name`: max. 32 characters.

Saved

Only with *Appearance* set in the *File > Configuration > Save Configuration* dialog box.

Example

```
WindowFont=-16,500,0,Courier
```

TipFilePos

Arguments

any integer, e.g., 236

Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

Saved

Always when saving a configuration file.

ShowTipOfDay

Description

Determines whether the *Tip of the Day* dialog box is shown at startup.

- 1: Show Tip of the Day
- 0: Show Tip of the Day only when accessed from the Help menu

Arguments

0/1

Saved

Always when saving a configuration file.

Options

Description

The currently active option string. This entry can be very long.

Arguments

current option string, e.g.: `-W2`

Saved

Only with *Options* set in the *File > Configuration > Save Configuration* dialog box.

EditorType

Description

This entry specifies which editor configuration is active:

- 0: global editor configuration (in the file `mcutools.ini`)
- 1: local editor configuration (the one in this file)
- 2: command line editor configuration, entry EditorCommandLine
- 3: DDE editor configuration, entries beginning with EditorDDE
- 4: CodeWarrior with COM. There are no additional entries.

For details, see also Editor Settings Dialog Box.

Arguments

0/1/2/3/4

Saved

Only with *Editor Configuration* set in the *File > Configuration > Save Configuration* dialog box.

EditorCommandLine

Description

Command line content to open a file. For details, see also Editor Settings Dialog Box.

Arguments

Command line, for `UltraEdit-32: “c:\Programs Files\IDM Software Solutions\UltraEdit-32\uedit32.exe %f -g%l,%c”`

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

EditorDDEClientName

Description

Name of the client for DDE editor configuration. For details, see also Editor Settings Dialog Box.

Arguments

client command, e.g., `“[open(%f)]”`

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

EditorDDETopicName

Description

Name of the topic for DDE editor configuration. For details, see also Editor Settings Dialog Box.

Arguments

topic name, e.g., “system”

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

EditorDDEServiceName

Description

Name of the service for DDE editor configuration. For details, see also Editor Setting dialog box.

Arguments

service name, e.g., “`system`”

Saved

Only with *Editor Configuration* set in the *File* > *Configuration* > *Save Configuration* dialog box.

Example

The example in Listing B.1 shows a typical layout of the configuration file (usually `project.ini`).

Listing B.1 Example of a project.ini File

```
[Editor]
Editor_Name=IDF
Editor_Exe=c:\Freescale\prog\idf.exe
Editor_Opts=%f -g%l,%c

[HC12_Assembler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.asm -w2
RecentCommandLine1=fibo.asm
CurrentCommandLine=fibo.asm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=c:\Freescale\prog\idf.exe %f -g%l,%c
```

Index