CodeWarrior Development Studio for Microcontrollers V10.x HC(S)08 Build Tools Reference Manual

Document Number: CWMCUS08CMPREF
Rev 10.6, 01/2014

Contents

Section number **Title** **Page**

Chapter 1
Overview

Chapter 2
Using Compiler

Chapter 3
Introduction

Chapter 4
Graphical User Interface

Chapter 5
Environment

Chapter 6
Files

Chapter 7
Compiler Options

Chapter 8
Compiler Predefined Macros

Chapter 9
Compiler Pragmas

Chapter 10
ANSI-C Frontend

Chapter 11
Generating Compact Code

Chapter 12
HC(S)08 Backend

Chapter 13
High-Level Inline Assembler for the HC(S)08

Chapter 14
ANSI-C Library Reference

Chapter 15
Library Files

Chapter 16
Special Features

Chapter 17
Library Structure

Chapter 18
Types and Macros in the Standard Library

Chapter 19
The Standard Functions

Chapter 20
Appendices

Chapter 21
Porting Tips and FAQs

Chapter 22
Global Configuration File Entries

Chapter 23
Local Configuration File Entries

Chapter 24
Known C++ Issues in the HC(S)08 Compilers

Chapter 25
Banked Memory Support

Chapter 26
Compiler Messages

Chapter 1
Overview

The HC(S)08 Build Tools Reference Manual for Microcontrollers describes the compiler used for the Freescale 8-bit Microcontroller Unit (MCU) chip series. This document consists of the following sections:

- Using Compiler : Describes how to run the compiler
- ANSI-C Library Reference : Describes how the compiler uses the ANSI-C library
- Appendices : Lists FAQs, global and local configuration file entries, and known C++ issues

NOTE

The technical notes and application notes are placed at the following location: `<CWInstallDir>\MCU\Help\PDF`, where, `CWInstallDir` is the directory in which the CodeWarrior software is installed.

This chapter covers the following topics:

- Accompanying Documentation
- Additional Information Resources

1.1 Accompanying Documentation

The **Documentation** page describes the documentation included in the *CodeWarrior Development Studio for Microcontrollers v10.x*. You can access the **Documentation** by:

- opening the `START_HERE.html` in `<CWInstallDir>\MCU\Help` folder,
- selecting **Help > Documentation** from the IDE's menu bar, or selecting the **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > Documentation** from the Windows taskbar.

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and debugger, refer to the *CodeWarrior Common Features Guide* in this folder: `<CWInstallDir>\MCU\Help\PDF`

1.2 Additional Information Resources

Refer to the documentation listed below for details about the programming languages:

- *American National Standard for Programming Languages - C*, ANSI/ISO 9899-1990 (see ANSI X3.159-1989, X3J11)
- *The C Programming Language*, second edition, Prentice-Hall 1988
- *C: A Reference Manual*, second edition, Prentice-Hall 1987, Harbison and Steele
- *C Traps and Pitfalls*, Andrew Koenig, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Nov. 1988, ISBN 0-201-17928-8
- *Data Structures and C Programs*, Van Wyk, Addison-Wesley 1988
- *How to Write Portable Programs in C*, Horton, Prentice-Hall 1989
- *The UNIX Programming Environment*, Kernighan and Pike, Prentice-Hall 1984
- *The C Puzzle Book*, Feuer, Prentice-Hall 1982
- *C Programming Guidelines*, Thomas Plum, Plum Hall Inc., Second Edition for Standard C, 1989, ISBN 0-911537-07-4
- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 1.1.0 (October 6, 1992), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 2.0.0 (July 27, 1993), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *System V Application Binary Interface*, UNIX System V, 1992, 1991 UNIX Systems Laboratories, ISBN 0-13-880410-9
- *Programming Microcontroller in C*, Ted Van Sickle, ISBN 1878707140
- *C Programming for Embedded Systems*, Kirk Zurell, ISBN 1929629044
- *Programming Embedded Systems in C and C ++*, Michael Barr, ISBN 1565923545
- *Embedded C*, Michael J. Pont, ISBN 020179523X

Chapter 2
Using Compiler

This section consists of the following chapters that describe the use and operation of the compiler:

- Introduction : Describes the compiler.
- Graphical User Interface : Describes the compiler's GUI.
- Environment : Describes the environment variables.
- Files : Describes how the compiler processes input and output files.
- Compiler Options : Describes the full set of compiler options.
- Compiler Predefined Macros : Lists all predefined macros of the compiler.
- Compiler Pragmas : Lists the available pragmas.
- ANSI-C Frontend : Describes the ANSI-C implementation.
- Generating Compact Code : Describes programming guidelines for the developer to produce compact and efficient code.
- HC(S)08 Backend : Describes code generator and basic type implementation, also hints about hardware-oriented programming (optimizations and interrupt functions) specific for HC(S)08.
- High-Level Inline Assembler for the HC(S)08 : Describes the HLI assembler for the HC(S)08 compiler.

Chapter 3
Introduction

This chapter describes the HC(S)08 Compiler that is part of the CodeWarrior Development Studio for Microcontrollers V10.x. The Compiler consists of a *Frontend*, which is language-dependent, and a *Backend,* which is hardware-dependent and generates object code specific to HC08 and HCS08. Chapters one and two describe the configuration and creation of projects that target HC08 and HCS08 microcontrollers. Chapter three describes the Compiler in greater detail.

NOTE

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

This chapter consists of the following topics.

3.1 Compiler Environment

Use the Compiler alone or as a transparent, integral part of the CodeWarrior Development Studio. Create and compile functional projects in minimal time using the Eclipse Integrated Development Environment (IDE), or configure the compiler and use it as a standalone application in a suite of the Build Tool utilities. The Build Tool utilities include the Linker, Assembler, ROM Burner, Simulator, and Debugger.

In general, a Compiler translates source code, such as C source code files (`*.c`) and header (`*.h`) files, into object code (`*.obj and *.o`) files for further processing by a Linker. The `*.c` files contain the programming code for the project's application. The `*.h` files either contain data specifically targeted to a particular microcontroller or are function-interface files. The Linker, under the command of a linker command file, uses the object-code file to directly generates absolute (`*.abs`) files. The Burner uses the `*.abs` files to produce S-record (`*.s19` or `*.sx`) files for programming the ROM.

For information about mixing assembly and C source code in a single project, refer to the topic, High-Level Inline Assembler for the HC(S)08.

The typical configuration of the Compiler is its association with a project directory and an editor. The project directory contains all of the environment files required to configure your development environment and editor allows to write or modify your source files.

NOTE

For information about the other Build Tools, refer to the User Guides located at `<CWInstallDir>\MCU\Help`, where `CWInstallDir` is the directory in which the CodeWarrior software is installed.

3.2 Designing Project

There are three methods of designing a project:

- Use the CodeWarrior IDE to coordinate and manage the entire project (Creating and Managing Project Using CodeWarrior IDE),
- Begin project creation with CodeWarrior IDE and use the standalone build tools to complete the project (Using Standalone Compiler), or
- Start from scratch, make your project configuration (`*.ini`) and layout files for use with the Build Tools, (Build Tools (Application Programs))

NOTE

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

CodeWarrior installation directory. The default location of this folder is: `C:\Freescale\CW MCU V10.x\MCU\prog`

3.3 Creating and Managing Project Using CodeWarrior IDE

You can create a Microcontrollers project and generate the basic project files using the **New Bareboard Project** wizard in the CodeWarrior IDE.

You can use the **CodeWarrior Projects** view in the CodeWarrior IDE to manage files in the project.

This section covers the following topics:

- Creating Project using CodeWarrior Project Wizard
- Analysis of Groups in CodeWarrior Projects View
- Analysis of Files in CodeWarrior Projects View

3.3.1 Creating Project using CodeWarrior Project Wizard

The steps below create an example Microcontrollers project that uses C language for its source code.

1. Select **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > CodeWarrior** from the Windows® Taskbar.

   The **Workspace Launcher** dialog box appears. The dialog box displays the default workspace directory. For this example, the default workspace is `workspace_MCU`.

2. Click **OK** to accept the default location. To use a workspace different from the default, click **Browse** and specify the desired workspace.

   The CodeWarrior IDE for Microcontrollers V10.x appears.

3. Select **File > New > Bareboard Project** from the IDE menu bar.

   The **Create an MCU Bareboard Project** page of the **New Bareboard Project** wizard appears.

4. Type the name of the project in the **Project name** text box. For this example, type `HCS08_project`. Click **Next**.

The **Devices** page displaying the supported Microcontrollers appears.

5. Select the desired CPU derivative for the project. For this example, select **S08 > HCS08G Family > MC9S08GT32**.

NOTE

Based on the derivative selected in the **Devices** page, the step numbering in the page title varies.

6. Click **Next**.

   The **Connections** page appears.

7. Check the option(s) to specify the hardware probe that you want to use to connect the workstation to the hardware target. By default, only the **P&E USB Multilink Universal [FX] / USB Multilink** is selected.
8. Click **Next**.

   The **Languages** page appears.

9. Select the programming language options you want to use. For this example, check the **C** checkbox.

NOTE

To enable the **Absolute Assembly** checkbox, clear the **C** , **C++** , and **Relocatable Assembly** checkboxes. This is because you cannot mix the absolute and relocatable assembly code in a program. Since the C and C++ compilers generate relocatable assembly, they must be cleared to allow the use of absolute assembly.

10. Click **Next**.

    The **Rapid Application Development** page appears.

11. Select the appropriate rapid application development options.
12. Click **Next**.

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

NOTE

As this project uses the **C** programming language, **Small**, **None** and **ANSI startup code** options are selected by default for the *memory model*, the *floating point supported* and the *level of startup code*, respectively.

13. Select the options appropriate for your project.
14. Click **Finish**.

The wizard automatically generates the startup and initialization files for the specific microcontroller derivative, and assigns the entry point into your ANSI-C project (the `main()` function). The `HCS08_project` project appears in the **CodeWarrior Projects** view in the Workbench window.

NOTE

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

By default, the project is not built. To do so, select **Project > Build Project** from the IDE menu bar. Expand the `HCS08_project : FLASH` tree control in the **CodeWarrior Projects** view to view its supporting directories and files.

NOTE

To configure the IDE, so that it automatically builds the project when a project is created, select **Window > Preferences** to open the **Preferences** window. Expand the **General** node and select **Workspace**. In the **Workspace** panel, check the **Build automatically** checkbox and click **OK**.

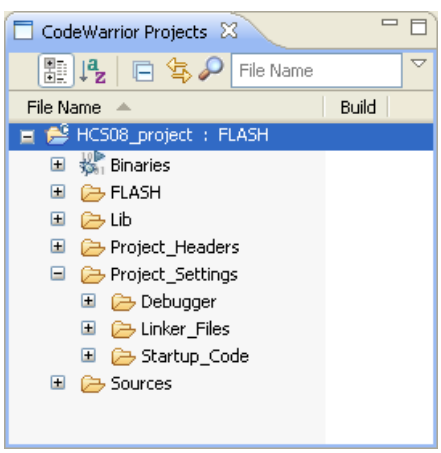

Figure 3-1. CodeWarrior Projects View - Displaying HCS08_project

NOTE

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

The view displays the logical arrangement of the files in the `HCS08_project` project directory. At this stage, you can safely close the project and reopen it later, if desired.

The following is the list of default groups and files displayed in the **CodeWarrior Projects** view.

- Binaries is a link to the generated binary (.abs) files.
- FLASH is the directory that contains all of the files used to build the application for HCS08_project. This includes the source files, the header files, the generated binary files, the makefiles that manage the build process, and the build settings.
- Lib is the directory that contains a C source code file that describes the chosen Microcontrollers derivative's registers and the symbols used to access them.
- Project_Headers is the directory that contains link to all of the header files (.h) that support the Microcontrollers derivative, plus any project-specific header files.
- Project_Settings group contains the Debugger, Linker_Files and the Startup_Code folders. The Linker_Files folder stores the linker command file (.prm) and the burner command file (.bbl). The Startup_Code folder has a C file that initializes the Microcontrollers stack and critical registers when the program launches. The Debugger folder consists of any initialization and memory configuration files that prepare the hardware target for debugging. It also stores the launch configuration used for the debugging session.
- Sources contains the source code files for the project. For this example, the wizard has created only main.c, which contains the main() function.

Examine the project folder that the IDE generates when you create the project. To do so, right-click the project's name (HCS08_project : FLASH) in the **CodeWarrior Projects** view, and select **Show In Windows Explorer**. Windows displays the Eclipse workspace folder, along with the project folder, HCS08_project, within it.

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

There are some files, .project, .cproject, and .cwGeneratedFilesetLog, that store critical information about the project's state. The **CodeWarrior Projects** view does not display these files, but they should not be deleted.

3.3.2 Analysis of Groups in CodeWarrior Projects View

In the **CodeWarrior Projects** view, the project files are distributed into five major groups, each with their own folder within the HCS08_project folder.

The default groups and their usual functions are:

- FLASH

  The HCS08 group contains all of the files that CodeWarrior uses to build the program. It also stores any files generated by the build process, such as any binaries (.o, .obj, and .abs), and a map file (.map). CodeWarrior uses this directory to manage the build

process, so you should not tamper with anything in this directory. This directory's name is based on the build configuration, so if you switch to a different build configuration, its name changes.

- `Lib`

  The `Lib` group contains the C-source code file for the specific Microcontrollers derivative. For this example, the `mc9s08gt32.c` file supports the `MC9S08GT32` derivative. This file defines symbols that you use to access the Microcontrollers registers and bits within a register. It also defines symbols for any on-chip peripherals and their registers. After the first build, you can expand this file to see all of the symbols that it defines.

- `Project_Headers`

  The `Project_Headers` group contains the derivative-specific header files required by the Microcontrollers derivative file in the `Lib` group.

- `Project_Settings`

  The `Project_Settings` group consists of the following sub-folders:

  - `Debugger`

    This group contains the files used to manage a debugging session. These are the debug launch configuration (`.launch`), a memory configuration file (`.mem`) for the target hardware, and any TCL script files (`.tcl`).

  - `Linker_Files`

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

  - `Startup_Code`

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

- `Sources`

  This group contains the C source code files. The **New Bareboard Project** wizard generates a default `main.c` file for this group. You can add your own source files to this folder. You can double-click on these files to open them in the IDE's editor. You can right-click the source files and select **Resource Configurations > Exclude from Build** to prevent the build tools from compiling them.

3.3.3 Analysis of Files in CodeWarrior Projects View

Expand the groups in the **CodeWarrior Projects** view to display all the default files generated by the **New Bareboard Project** wizard.

The wizard generates following three C source code files, located in their respective folders in the project directory:

- `main.c`,

  located in `<project_directory>\Sources`

- `start08.c`, and

  located in `<project_directory>\Project_Settings\Startup_Code`

- `mc9s08gt32.c`

  located in `<project_directory>\Lib`

At this time, the project should be configured correctly and the source code should be free of the syntactical errors. If the project has been built earlier, you should see a link to the project's binary files, and the `HCS08` folder in the **CodeWarrior Projects** view.

To understand what the IDE does while building a project, clean the project and build the project again:

1. Select **Project > Clean** *from the IDE menu bar.*

   The **Clean** dialog box appears.

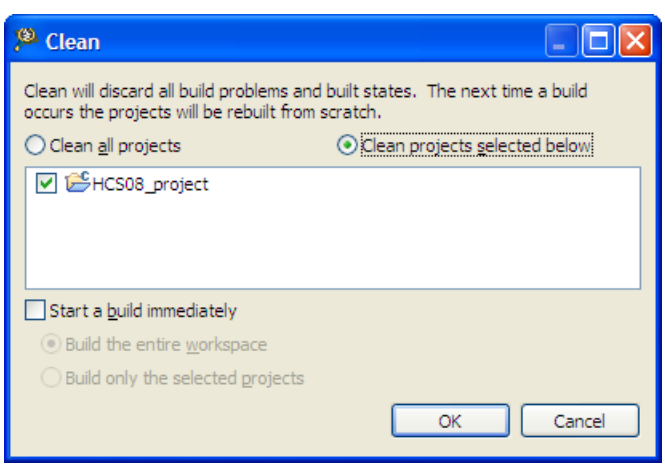

Figure 3-2. Clean Dialog Box

2. Select the **Clean projects selected below** option and check the project you want to build again.

3. Click **OK** .

   The Binaries link disappears, and the FLASH folder is deleted.

4. To build the project, right-click the project and select **Build Project** .

   The **Console** view displays the statements that direct the build tools to compile and link the project. The Binaries link appears, and so does the FLASH folder.

During a project build, the C source code is compiled, the object files are linked together, and the CPU derivative's ROM and RAM area are allocated by the linker according to the settings in the linker command file. When the build is complete, the FLASH folder contains the HCS08_project.abs file.

The Linker Map file, HCS08_project.map, indicates the memory areas allocated for the program.

To examine the source file, main.c, do the following:

- Double click the main.c file in the Sources group.

  The IDE opens the main.c file in the editor area.

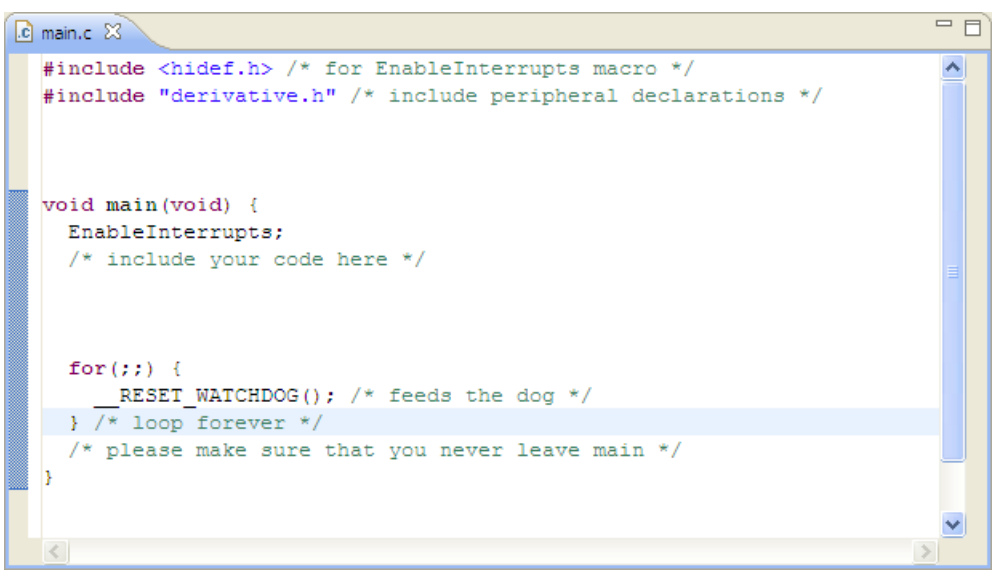

Figure 3-3. Default main.c File

Use the integrated editor to write your C source files (*.c and *.h) and add them to your project. During development, you can test your source code by building and simulating/ debugging your application.

3.4 Using Standalone Compiler

You can use the HC(S)08 Compiler as a standalone compiler. In this section, we will create a new project directory using some of the previously created files, and build the application.

NOTE

Although it is possible to create and configure a project from scratch using the standalone Build Tools, it is easier and faster to create, configure, and manage your project using the CodeWarrior Projects wizard.

This section covers the following topics:

3.4.1 Configuring the Compiler

Build tools use a *tool directory* for configuring and locating their generated files. For the Microcontrollers V10.x build tools, the tool directory is the `prog` folder, and is located in the `<CWInstallDir>\MCU` folder, where `CWInstallDir` is the directory in which the CodeWarrior software is installed.

Build tools require information from the configuration files. There are two types of configuration data:

- Global

  Global data is common to all build tools and projects, and includes data for each build tool, such as the directory path for recent project listings. All tools may store some global data in a `mcutools.ini` file. Each tool looks for this file in the tool directory first, then in the Microsoft Windows installation directory (for example, `C:\WINDOWS`). The following listing shows the locations for the global configuration file.

  Listing: Locations for Global Configuration File

```
<CWInstallDir>\MCU\prog\mcutools.ini - #1 priority
C:\WINDOWS\mcutools.ini - used if there is no mcutools.ini file above
```

The Compiler uses the initialization file in the `prog` folder if you start the tool from the `prog` folder.

That is, `<CWInstallDir>\MCU\prog\mcutools.ini`, where, `CWInstallDir` is the directory in which the CodeWarrior software is installed.

The Compiler uses the initialization file in the `Windows` directory if the tool is started outside the `prog` folder. That is, `C:\WINDOWS\mcutools.ini`

NOTE

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

- Local

  Local data files can be used by any build tool for a particular project. These files are typically stored in a project folder. For information about local configuration file entries, refer to the Local Configuration File Entries in the Appendices.

To configure the Compiler, proceed as follows:

1. Double-click `chc08.exe` file, which is located in the `<CWInstallDir>\MCU\prog` folder.

   The **HC08 Compiler** window appears.

Figure 3-4. HC08 Compiler Window

2. Read the tips, if you want to, and click **Close**.

   The **Tip of the Day** dialog box closes.

3. Select **File > New > Default Configuration** from the HC08 Compiler menu bar.

NOTE

You can also use an existing configuration file. Select **File > Load Configuration** to open the existing configuration file.

4. Select **File > Save Configuration** (or **Save Configuration As**) from the HC08 Compiler menu bar.

   The **Saving Configuration as** dialog box appears.

5. Browse and select the folder where you want to save the configuration. In this example, folder `X15` is used to save the configuration.

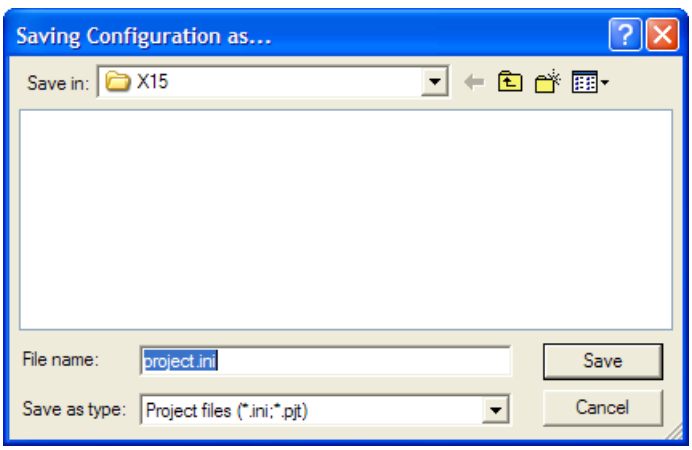

Figure 3-5. Saving Configuration as Dialog Box

6. Click **Save**.

   The folder becomes the project directory. The project directory folder is often referred as *current directory*.

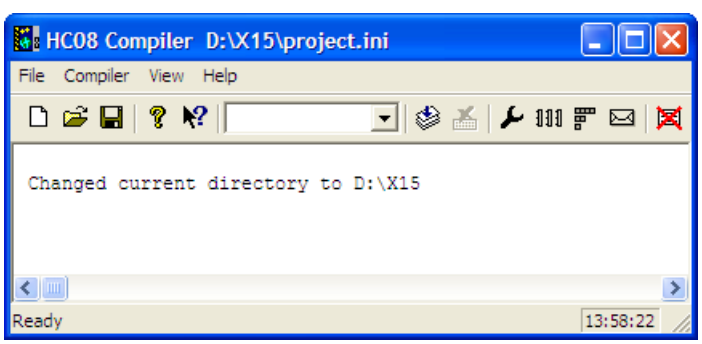

Figure 3-6. Compiler's Current Directory Switches to your Project Directory

The project directory folder now contains the newly-created `project.ini` configuration file. The project configuration file contains a copy of the *[CHC08_Compiler]* section of the `project.ini` file located in the `prog` folder. If a build tool modifies any project option, the changes are specified in appropriate section of the project configuration file.

NOTE

Option changes made to the `project.ini` file in the `prog` folder apply to all projects. Recall that this file stores the global tool options.

7. Select **Compiler > Options > Options** from the HC08 Compiler menu bar.

   The **HC08 Compiler Option Settings** dialog box appears.

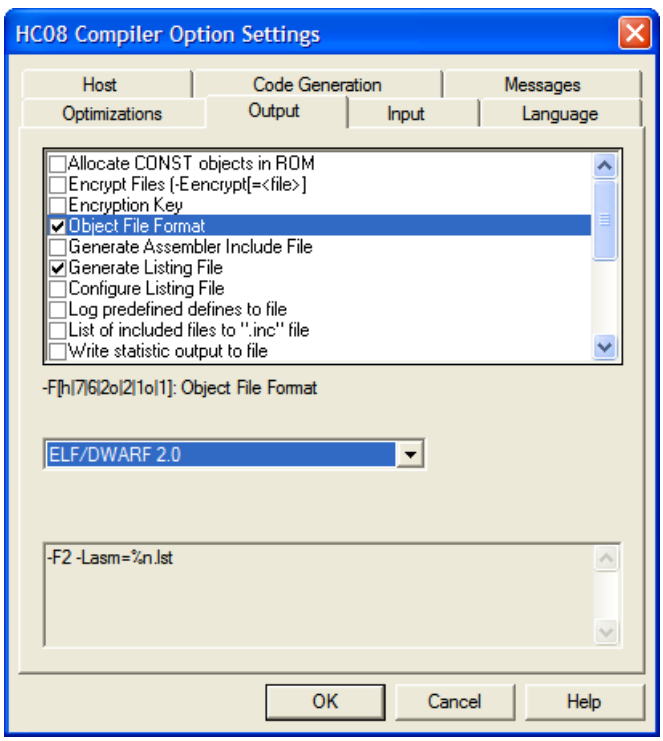

Figure 3-7. HC08 Compiler Option Settings Dialog Box

8. Select the **Output** tab.
9. Check the **Generate Listing File** *checkbox*.
10. Check the **Object File Format** checkbox and select **ELF/DWARF 2.0** from the drop-down list that appears for the **Object File Format** checkbox.
11. Click **OK**.

    The **HC08 Compiler Option Settings** dialog box closes.

12. Save the changes to the configuration by either:

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

The compiler is configured.

NOTE

For more information about configuring the Compiler options, refer to the Graphical User Interface chapter.

3.4.2 Selecting Input Files

The project does not contain any source code files at this point. Copy and paste the C source code (`*.c`) and header (`*.h`) files from the previous project, `HCS08_project`, to save time.

1. Copy the `Sources` folder from the `HCS08_project` folder and paste it into the `X15` project folder.
2. Copy the `Project_Headers` folder from the `HCS08_project` folder and paste it into the `X15` project folder.
3. Copy and paste the `HCS08_project` project's `Lib` folder into the `X15` project folder.
4. In the `X15` folder, create a new folder `Project_Settings`. Copy and paste the `Startup_Code` folder from the `HCS08_project\Project_Settings` folder into the `X15\Project_Settings` folder.

Now there are five items in the `X15` project folder, as shown in the figure below:

- the `project.ini` configuration file
- the `Sources` folder, which contains the `main.c` source file
- the `Project_Headers` folder, which contains:
  - the `derivative.h` header file, and
  - the `mc9s08gt32.h` header file
- the `Lib` folder, which contains the `mc9s08gt32.h` source file
- the `Project_settings\Startup_Code` folder, which contains the start up and initialization file, `start08.c`

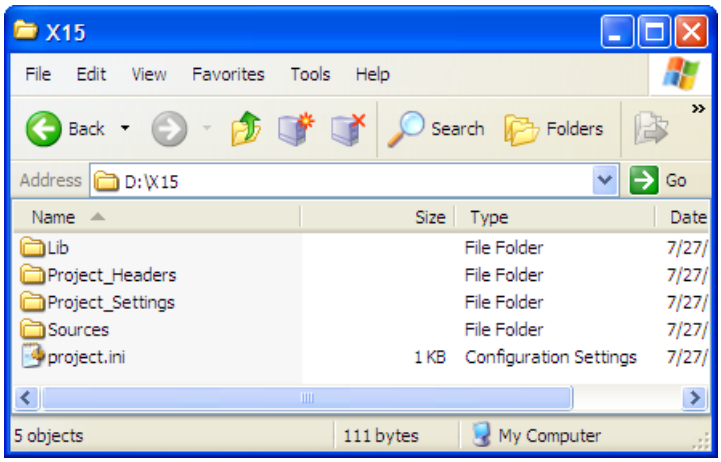

Figure 3-8. Folders and Files in X15 Project Folder

3.4.3 Compiling C Source Code Files

Now compile the `start08.c` file.

1. Select **File > Compile** from the HC08 Compiler menu bar.

   The **Select File to Compile** dialog box appears.

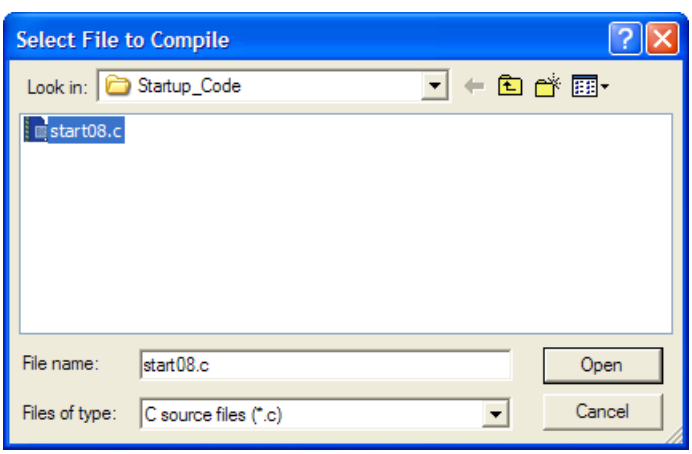

Figure 3-9. Select File to Compile Dialog Box

2. Browse to the `Project_settings\Startup_Code` folder in the project directory.
3. Select the `start08.c` file.
4. Click **Open**.

   The `start08.c` file begins compiling, as shown in the following figure. In this case, the file fails to compile.

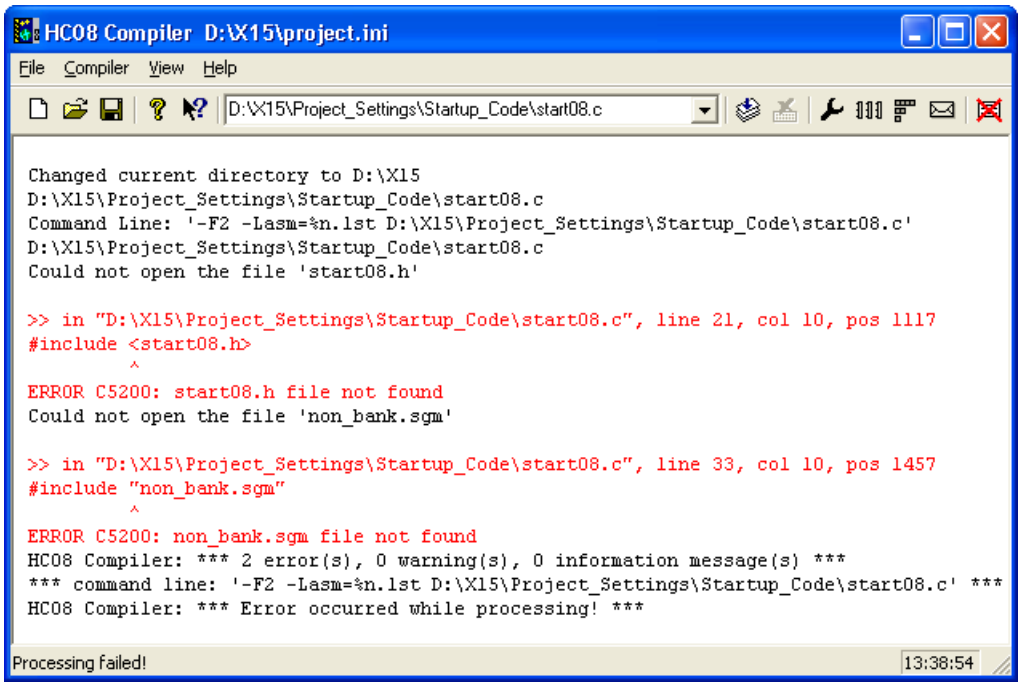

Figure 3-10. Results of Compiling start08.c File

The **HC08 Compiler** window provides information about the compilation process, and generates the error messages if the compilation fails. In this case, the `C5200: `FileName' file not found` error message appears twice, once for `start08.h` and once for `non_bank.sgm`.

5. Right-click the text above the error message.

   A context menu appears.

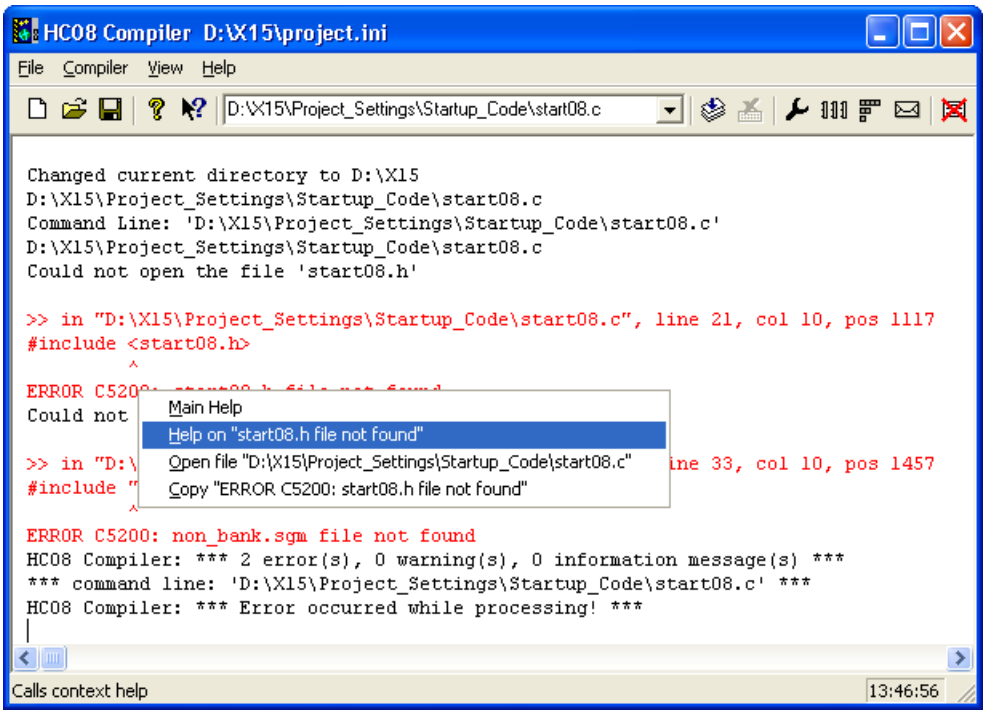

Figure 3-11. HC08 Compiler Window - Context Menu

6. Select **Help on "<FileName> file not found"** from the context menu.

   The **C5200: `FileName' file not found** error message appears.

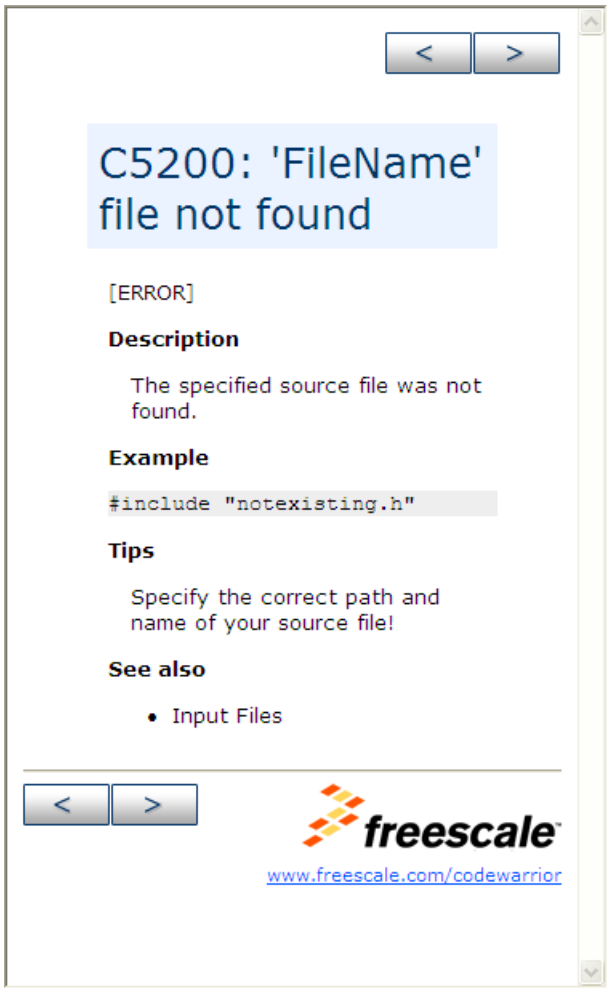

Figure 3-12. C5200 Error Message Help

The **Tips** section in the help for the *C5200 error* states that the correct paths and names for the source files must be specified. The following folder contains both the missing files, `start08.h` and `non_bank.sgm`:

```
<CWInstallDir>\MCU\lib\hc08c\include
```

NOTE

The `#include` preprocessor directive for the `start08.h` file and the `non_bank.sgm` file appears on line 21 and 31 of the `start08.c` file, respectively.

To resolve the error, modify the compiler configuration so that it can locate the missing files.

1. Select **File > Configuration** from the HC08 Compiler menu bar.

   The **Configuration** dialog box appears.

2. Select the **Environment** tab in the **Configuration** dialog box.
3. Select **Header File Path** and click [...] button.

   The **Browse for Folder** dialog box appears.

NOTE

The environment variable associated with the **Header File Path** is the `LIBPATH` or LIBRARYPATH: `include <File>' Path variable. The Compiler uses a hierarchy to search for files. Refer to the File Processing.

4. In the **Browse for Folder** dialog box, navigate to the missing `start08.h` file in the `<CWInstallDir>\MCU\lib\hc08c\include` folder, where `CWInstallDir` is the directory in which the CodeWarrior software is installed.

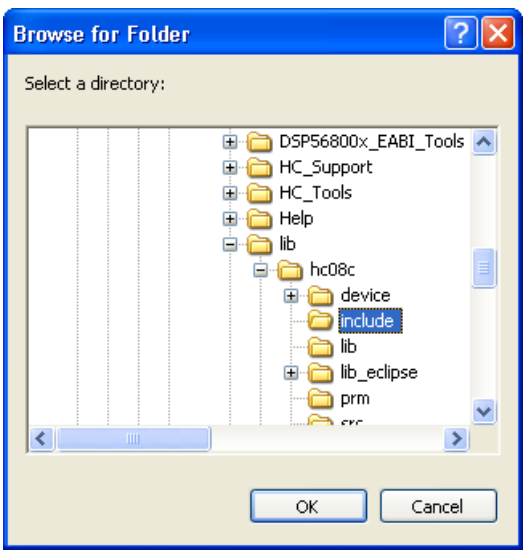

Figure 3-13. include Subfolder Containing start08.h and non_bank.sgm Files

5. Click **OK**.
6. Click **Add**.

   The specified path appears in the lower panel.

7. Click **OK**.

   An asterisk (*) now appears in the HC08 Compiler window's title bar, indicating that the configuration file contains unsaved changes.

8. Click **Save** on the toolbar to save the configuration modifications. Alternatively, select **File > Save Configuration** from the HC08 Compiler menu bar.

NOTE

If you do not save the configuration, the compiler reverts to the last-saved configuration when the program is

relaunched. The asterisk (*) disappears when the file is saved.

Now that you have specified the paths to the missing files, you can try again to compile the `start08.c` file.

1. Select **File > Compile** from the HC08 Compiler menu bar.
2. Navigate to the `Project_settings\Startup_Code` folder in the project directory.
3. Select `start08.c`.
4. Click **Open**.

The HC08 Compiler window indicates successful compilation of the `start08.c` file and displays following results:

- The `start08.c` file involved the use of four header files.
- The Compiler generated an assembler listing file (`start08.lst`) for the `start08.c` file.
- The Compiler generated an object file (`start08.o`) in the `Sources` folder, using the ELF/DWARF 2.0 format.
- The Code Size was 149 bytes.
- The compiler created six global objects and the Data Size was 6 bytes in RAM.
- There were 0 errors, 0 warnings, and 0 information messages.

Tip

Clear the Compiler window at any time by selecting **View > Log > Clear Log** from the HC08 Compiler menu bar.

Now compile the `main.c` file:

1. Select **File > Compile** from the HC08 Compiler menu bar.
2. Navigate to the `Sources` folder.
3. Select the `main.c` file.
4. Click **Open**.

The C source code file, `main.c`, fails to compile, as the compiler can locate only three of the four header files required. It could not find the `derivative.h` header file and generates another C5200 error message.

The `derivative.h` file is in the `Project_Headers` folder in the `X15` project folder, so add another header path to resolve the error.

1. Select **File > Configuration** from the HC08 Compiler menu bar.

   The **Configuration** dialog box appears.

2. Select the **Environment** tab in the **Configuration** dialog box.
3. Select **Header File Path** and click the [...] button.

The **Browse for Folder** dialog box appears.

4. In the **Browse for Folder** dialog box, navigate and select the `Project_Headers` folder.
5. Click **OK**.

   The **Browse for Folder** dialog box closes.

6. Click **Add**.

   The selected path appears in the lower panel.

7. Click **OK**.

   If there is no other missing header file included in the `derivative.h` file, you are ready to compile the `main.c` file.

8. Select **File > Compile** from the HC08 Compiler window menu bar.
9. Select `main.c` in the `Sources` folder. You can also select a previously compiled C source code file.
10. Click **Open** to compile the file.

    The message "`*** 0 error(s),`" indicates that `main.c` compiles without errors. Save the changes in the project configuration.

The compiler places the object file in the `Sources` folder, and generates output listing files in the project folder. The binary object files and the input modules have identical names except for the extension used. Any assembly output files generated for each C source code file is similarly named.

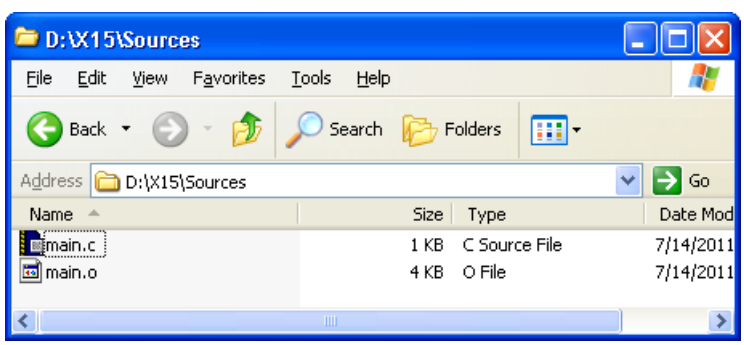

Figure 3-14. Project Directory after Successful Compilation

At this time, only two of the three C source code files have been compiled. Locate the remaining C source code file, `MC9S08GT32.c`, in the `Lib` folder of the current directory, `X15`.

The compiler places the object-code files it generates in the same folder that contains the C source code files. However, you can also specify a different location for the object-code files.

To redirect the object-code file for the `MC9S08GT32.c` file to another folder, modify the compiler configuration so that when the `MC9S08GT32.c` file is compiled, the object code file goes into a different folder. For more information, refer to the OBJPATH: Object File Path).

1. Using Windows Explorer, create a new folder in the current directory and name it `ObjectCode`.
2. Select **File > Configuration** from the HC08 Compiler menu bar.
3. Select the **Environment** tab and select **Object Path**.
4. Click **Browse**.

   The **Browse for Folder** dialog box appears.

5. In the **Browse for Folder** dialog box, browse for and select the `ObjectCode` folder.
6. Click **OK**.

   The **Browse for Folder** dialog box closes.

7. Click **Add**.

   The new object path appears in the lower pane.

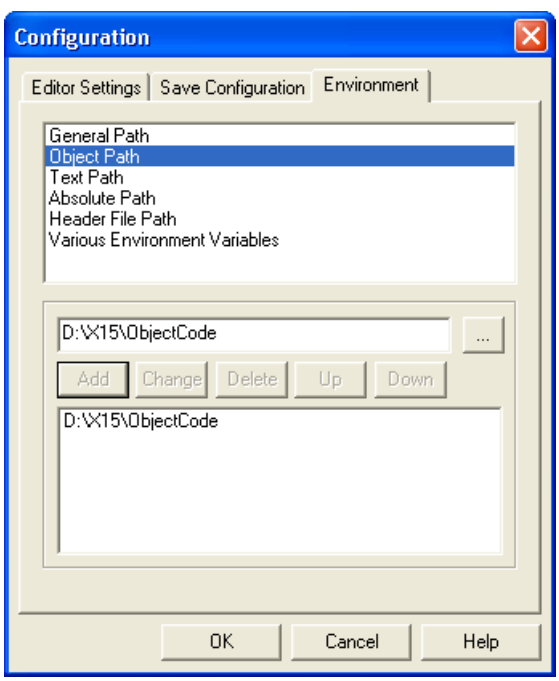

Figure 3-15. Adding OBJPATH

8. Click **OK**.

   The **Configuration** dialog box closes.

9. Press **Ctrl + S** to save the settings.
10. Select **File > Compile** from the HC08 Compiler menu bar.

The **Select File to Compile** dialog box appears.

11. Browse for the C source code file in the `Lib` subfolder of the project folder.

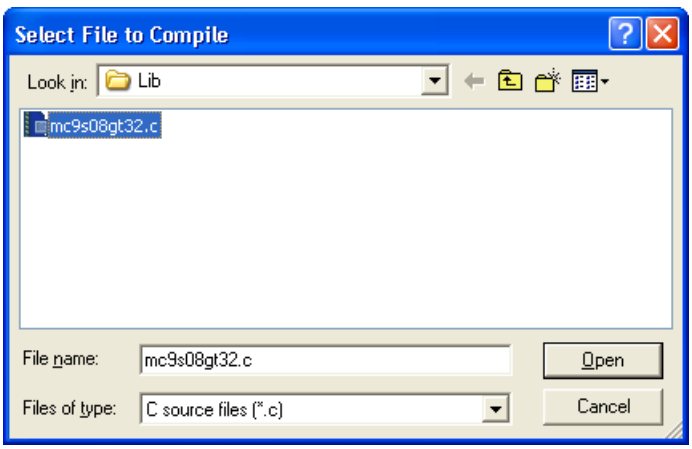

Figure 3-16. Compiling Source File in Lib Folder

12. Click **Open**.

The selected file compiles.

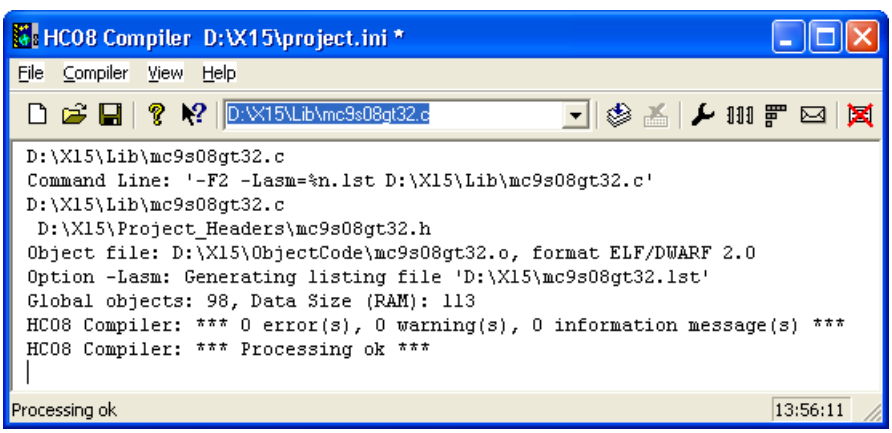

Figure 3-17. Successful Compilation of File in Lib Folder

The Compiler log states that the object code file, `MC9S08GT32.o`, is created in the `ObjectCode` folder, as specified. Save the project configuration again in case you wish to recompile any of the C source code files in future.

Tip

Move the other two object code files to the `ObjectCode` folder so that all object code files for the project are in the same place. This makes project management easier.

The haphazard builds of this project are intentionally designed to illustrate what happens if paths are not configured properly while compiling a project using the Compiler tool. The header files may be included by either C source or other header files. The `lib` folder in the CodeWarrior installation directory contains derivative-specific header and other files for Microcontrollers projects.

When you build another project with the Build Tool Utilities, make sure that the settings for the input and output files are done in advance.

Now that the project's object code files are available, you can use the linker build tool, `linker.exe`, together with an appropriate `*.prm` file, to link these object-code files with the library files and create an `*.abs` executable output file.

Refer to the *Linker* section in the *Build Tool Utilities Manual* for details. However, the project set up and configuration is faster and easier using the **New Bareboard Project** wizard.

3.5 Build Tools (Application Programs)

The standalone build tools (application programs) can be found in the `\prog` folder, which is located within the folder where the CodeWarrior software is installed.

```
<CWInstallDir>\MCU\prog
```

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

The following table lists the build tools used for C programming and generating an executable program.

Table 3-1. Build Tools

Build Tool	Description
`ahc 08.exe`	Freescale HC(S)08/RS08 assembler
`burner.exe`	Batch and interactive burner application that generates S-records
`chc 08.exe`	Freescale HC(S)08 compiler
`decoder.exe`	Decoder tool to generate assembly listings
`libmaker.exe`	Librarian tool to build libraries
`linker.exe`	Link tool to build applications (absolute files)
`maker.exe`	Make tool to rebuild the program automatically
`piper.exe`	Utility to redirect messages to stdout

NOTE

There may be some additional tools in your \prog folder or you may find some tools missing, depending on your license configuration.

3.6 Startup Command-Line Options

Startup command-line options are special tool options that are specified at tool startup, that is while launching the tool. They cannot be specified interactively.

Some of these options open special dialog boxes, such as the Configuration and Message dialog boxes, for a build tool.

The following table lists the Startup command-line options and their description:

Table 3-2. Startup Command-line Options

Startup Command-line Option	Description
-Prod	Specifies the current project directory or file. For more information, refer to the topic, -Prod: Specify Project File at Startup.
ShowOptionDialog	Opens the Option dialog box.
ShowMessageDialog	Opens the message dialog box.
ShowConfigurationDialog	Opens the **File > Configuration** dialog box.
ShowBurnerDialog	Opens the **Burner** dialog box.
ShowSmartSliderDialog	Opens the **CompilerSmart Control** dialog box. This is compiler option.
ShowAboutDialog	Opens the **About** dialog box.

NOTE

For more information on the build tool dialog boxes, refer to the Graphical User Interface chapter.

Listing: Example of -Prod Command-Line Option

```
C:\Freescale\CW MCU V10.x\MCU\prog>linker.exe -
Prod=C:\Freescale\demo\myproject.pjt
```

Listing: Example of Storing Options in Current Project Settings File

```
C:\Freescale\CW MCU V10.x\MCU\prog>linker.exe -
ShowOptionDialog -Prod=C:\demos\myproject.pjt
```

3.7 Highlights

The CodeWarrior build tools provide the following features:

- Powerful User Interface
- Online Help
- Flexible Type Management
- Flexible Message Management
- 32-bit Application
- Support for Encrypted Files
- High-Performance Optimizations
- Conforms to ANSI/ISO 9899-1990

3.8 CodeWarrior Integration of Build Tools

All required CodeWarrior plug-ins are installed together with the Eclipse IDE. The program that launches the IDE with the CodeWarrior tools, `cwide.exe`, is installed in the `eclipse` directory (by default, `C:\Freescale\CW MCU V10.x\eclipse`). The plug-ins are installed in the `eclipse\plugins` directory.

This section covers the following topics:

- Combined or Separated Installations
- HCS08 Compiler Build Settings Panels
- CodeWarrior Tips and Tricks

3.8.1 Combined or Separated Installations

The installation script enables you to install several CPUs along one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

NOTE

It is possible to have separate installations on one machine. There is only one point to consider: The IDE uses COM files, and for COM the IDE installation path is written into the Windows Registry. This registration is done in the installation setup. However, if there is a problem with the COM registration

using several installations on one machine, the COM registration is done by starting a small batch file located in the `bin` (by default `C:\Freescale\CW MCU V10.x\MCU\bin`) directory. To do this, start the `regservers.bat` batch file.

3.8.2 HCS08 Compiler Build Settings Panels

The following sections describe the settings panels that configure the HC(S)08 Compiler build tool options. These panels are part of the project's build properties settings, which are managed in the **Properties** window. To access these panels, proceed as follows:

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

   The **Properties for <project>** dialog box appears.

3. Expand the **C/C++ Build** tree control and select **Settings**.

   The various settings for the build tools are displayed in the right panel. If not, select the **Tool Settings** tab.

The options are grouped by tool, such as **Messages** , **Host** , **General** , **Disassembler** , **Linker** , **Burner** , **HCS08Compiler** , **HCS08 Assembler** and **Preprocessor** options. Depending on the build properties you wish to configure, select the appropriate option in the **Tool Settings** tab page.

NOTE

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

The following listed are the build properties panels for the HC(S)08 Compiler.

Table 3-3. Build Properties Panels for HC(S)08 Compiler

Build Tool	Build Properties Panels
HCS08 Compiler	HCS08 Compiler > Output
	HCS08 Compiler > Output > Configure Listing File
	HCS08 Compiler > Output > Configuration for list of included files in make format
	HCS08 Compiler > Input
	HCS08 Compiler > Language
	HCS08 Compiler > Language > CompactC++ features

Table continues on the next page...

Table 3-3. Build Properties Panels for HC(S)08 Compiler (continued)

Build Tool	Build Properties Panels
	HCS08 Compiler > Host
	HCS08 Compiler > Code Generation
	HCS08 Compiler > Messages
	HCS08 Compiler > Messages > Disable user messages
	HCS08 Compiler > Preprocessor
	HCS08 Compiler > Type Sizes
	HCS08 Compiler > General
	HCS08 Compiler > Optimization
	HCS08 Compiler > Optimization > Tree optimizer
	HCS08 Compiler > Optimization > Optimize Library Function
	HCS08 Compiler > Optimization > Branch Optimizer
	HCS08 Compiler > Optimization > Peephole Optimization

3.8.2.1 HCS08 Compiler

Use this panel to specify the command, options, and expert settings for the HC(S)08 compiler. Additionally, the **HCS08 Compiler** tree control includes the general, include file search path settings. The following figure shows the HC(S)08 **Compiler** settings.

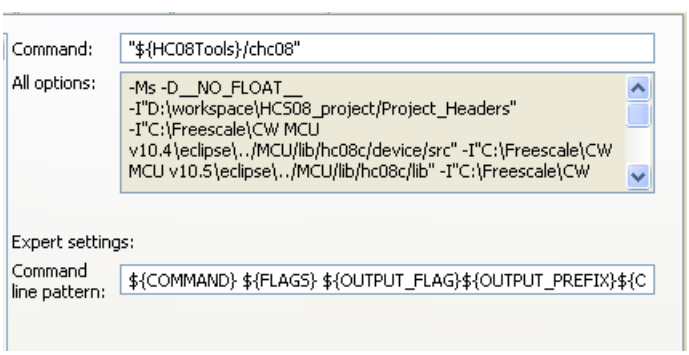

Figure 3-18. Tool Settings - HCS08 Compiler

The following table lists the **HSC08 Compiler** panel options for HC(S)08 Compiler.

Table 3-4. Tool Settings - HCS08 Compiler Options

Option	Description
Command	Shows the location of the linker executable file. Default value is: `"${HC08Tools}/chc08"`
All options	Shows the actual command line the compiler will be called with.

Table continues on the next page...

Table 3-4. Tool Settings - HCS08 Compiler Options (continued)

Option	Description
Expert Settings Command line pattern	Shows the expert settings command line parameters; default is `${COMMAND} ${FLAGS}${OUTPUT_FLAG}${OUTPUT_PREFIX}${OUTPUT}${INPUTS}`

3.8.2.2 HCS08 Compiler > Output

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

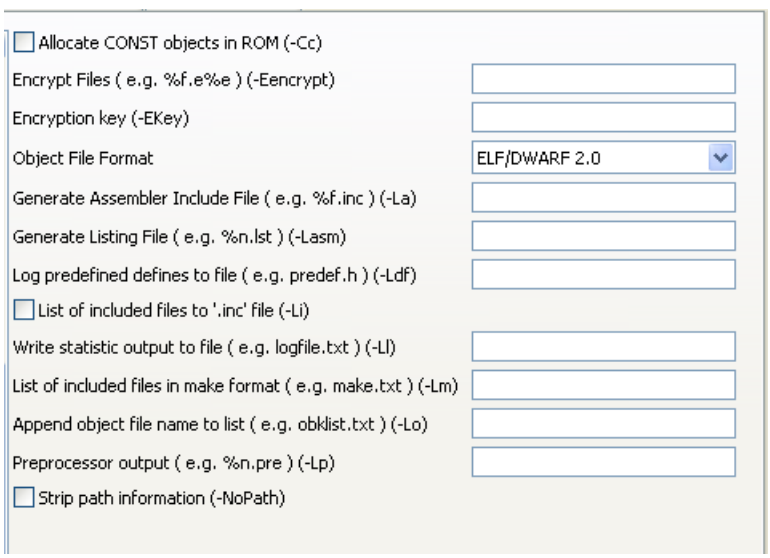

Figure 3-19. Tool Settings - HCS08 Compiler > Output

The following table lists the **Output** panel options for HC(S)08 compiler.

Table 3-5. Tool Settings - HCS08 Compiler > Output Options

Option	Description
Allocate `CONST` objects in ROM (`-Cc`)	Refer to the -Cc: Allocate Const Objects into ROM topic.
Encrypt Files (e.g. %f.e%e) (`-Eencrypt`)	Refer to the -Eencrypt: Encrypt Files topic.
Encryption key (`-EKey`)	Refer to the -Ekey: Encryption Key topic.
Object File Format	Refer to the -F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format topic.
General Assembler Include File (e.g. `%f.inc`) (`-La`)	Refer to the -La: Generate Assembler Include File topic.
Generate Listing File (e.g. `%n.lst`) (`-Lasm`)	Refer to the -Lasm: Generate Listing File topic.

Table continues on the next page...

Table 3-5. Tool Settings - HCS08 Compiler > Output Options (continued)

Option	Description
Log predefined defines to file (e.g. `predef.h`) (`-Ldf`)	Refer to the -Ldf: Log Predefined Defines to File topic.
List of included files to ` .inc' file (`-Li`)	Refer to the -Li: List of Included Files to ".inc" File topic.
Write statistic output to file (e.g. `logfile.txt`) (`-Ll`)	Refer to the -Ll: Write Statistics Output to File topic.
List of included files in make format (e.g. `make.txt`) (`-Lm`)	Refer to the -Lm: List of Included Files in Make Format topic.
Append object file name to list (e.g. `obklist.txt`) (`-Lo`)	Refer to the -Lo: Append Object File Name to List (enter [<files>]) topic.
Preprocessor output (e.g. `%n.pre`) (`-Lp`)	Refer to the -Lp: Preprocessor Output topic.
Strip path information (`-NoPath`)	Refer to the -NoPath: Strip Path Info topic.

3.8.2.3 HCS08 Compiler > Output > Configure Listing File

Use this panel to configure the listing files for HC(S)08 Compiler. The following figure shows the **Configure Listing File** panel.

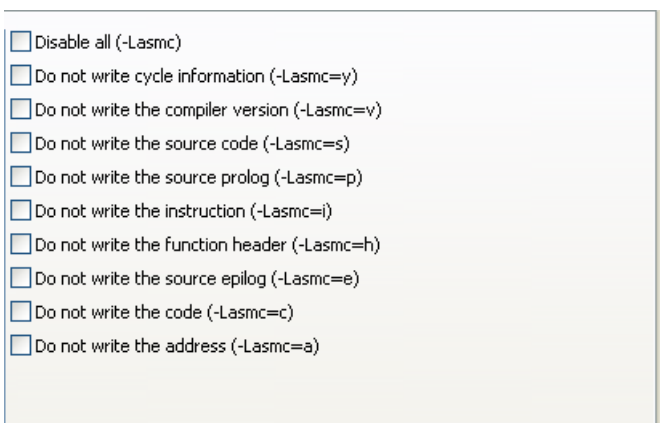

Figure 3-20. Tool Settings - HCS08 Compiler > Output > Configure Listing File

NOTE

For information about the description of the options appearing in the **Configure Listing File** panel for HC(S)08 Compiler, refer to the -Lasmc: Configure Listing File topic.

3.8.2.4 HCS08 Compiler > Output > Configuration for list of included files in make format

Use this panel to configure the list of included files in make format for HC(S)08 Compiler to generate the output.

The following figure shows the Configuration for list of included files in make format panel.

NOTE

For information about the description of the options appearing in the **Configuration for list of included files in make format** panel for HC(S)08 Compiler, refer to the -LmCfg: Configuration for List of Included Files in Make Format (option -Lm) topic.

3.8.2.5 HCS08 Compiler > Input

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

The IDE first looks for an include file in the current directory, or the directory that you specify in the `INCLUDE` directive. If the IDE does not find the file, it continues searching the paths shown in this panel. The IDE keeps searching paths until it finds the `#include` file or finishes searching the last path at the bottom of the Include File Search Paths list. The IDE appends to each path the string that you specify in the `INCLUDE` directive.

NOTE

The IDE displays an error message if a header file is in a different directory from the referencing source file. Sometimes, the IDE also displays an error message if a header file is in the same directory as the referencing source file.

For example, if you see the message `Could not open source file myfile.h`, you must add the path for `myfile.h` to this panel. The following figure shows the **Input** panel.

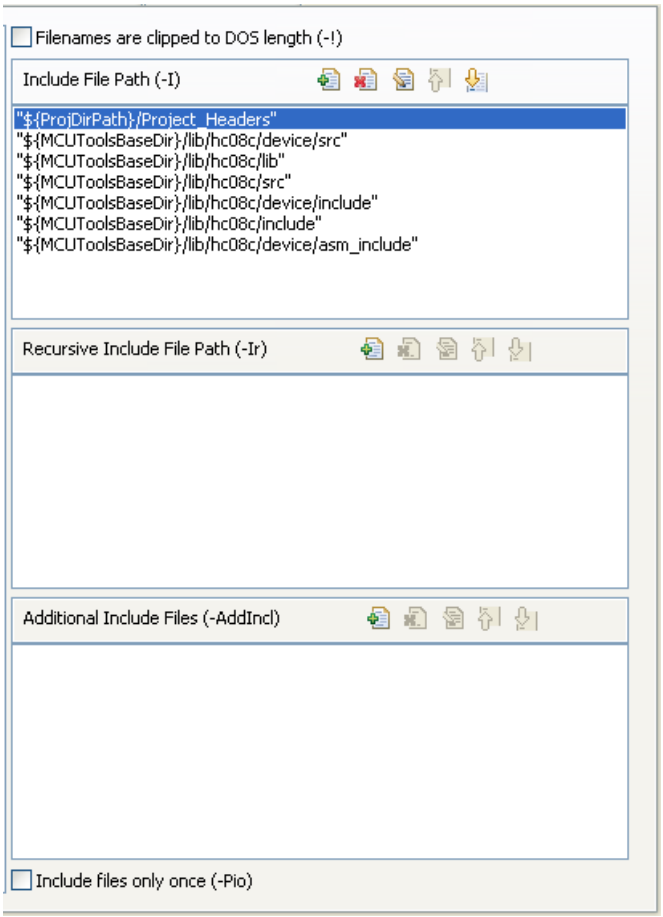

Figure 3-21. Tool Settings - HCS08 Compiler > Input

The following table lists the **Input** panel options for HC(S)08 Compiler.

Table 3-6. Tool Settings - HCS08 Compiler > Input Options

Option	Description
Filenames are clipped to DOS length (`-!`)	Refer to the -!: Filenames are clipped to DOS Length topic.
Include File Path (`-I`)	Refer to the -I: Include File Path topic.
Additional Include Files (`-AddIncl`)	Refer to the -AddIncl: Additional Include File topic.
Include files only once (`-Pio`)	Refer to the -Pio: Include Files Only Once topic.

The following table lists and describes the toolbar buttons that help work with the file paths.

Table 3-7. Include File Path (-I) Toolbar Buttons

Button	Description
	**Add** - Click to open the **Add directory path** dialog box and specify location of the library you want to add.
	**Delete** - Click to delete the selected library path.

Table continues on the next page...

Table 3-7. Include File Path (-I) Toolbar Buttons (continued)

Button	Description
	**Edit** - Click to open the **Edit directory path** dialog box and update the selected path.
	**Move up** - Click to move the selected path one position higher in the list.
	**Move down** - Click to move the selected path one position lower in the list.

The following figure shows the **Add directory path** dialog box.

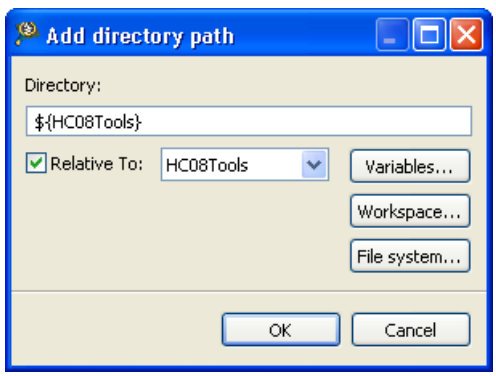

Figure 3-22. Tool Settings - HCS08 Compiler > Input - Add Directory Path Dialog Box

The following figure shows **Edit directory path** dialog box.

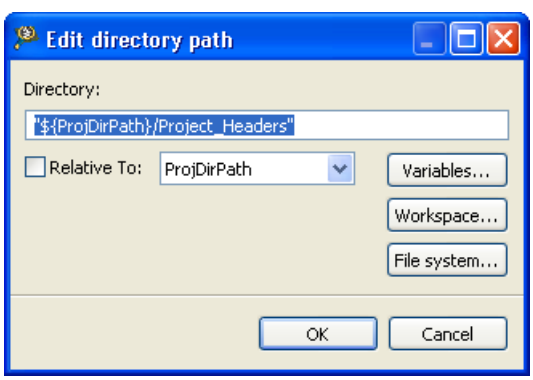

Figure 3-23. Tool Settings - HCS08 Compiler > Input - Edit Directory Path Dialog Box

The buttons in the **Add directory path** and **Edit directory path** dialog boxes help work with the paths.

- **OK-** Click to *confirm* the action and exit the dialog box.
- **Cancel-** Click to *cancel* the action and exit the dialog box.
- **Workspace -** Click to display the **File selection** dialog box and specify the path. The resulting path, relative to the workspace, appears in the appropriate list.

- **File system -** Click to display the **Browse For Folder** dialog box and specify the path. The resulting path appears in the appropriate list.
- **Variables -** Click to display the **Select build variable** dialog box. The resulting path appears in the appropriate list.

The following table lists and describes the toolbar buttons that help work with the search paths.

Table 3-8. Additional Include Files (-AddIncl) Toolbar Buttons

Button	Description
	**Add** - Click to open the **Add file path** dialog box and specify location of the library you want to add.
	**Delete** - Click to delete the selected library path.
	**Edit** - Click to open the **Edit file path** dialog box and update the selected path.
	**Move up** - Click to move the selected path one position higher in the list.
	**Move down** - Click to move the selected path one position lower in the list.

The following figure shows **Add file path** dialog box.

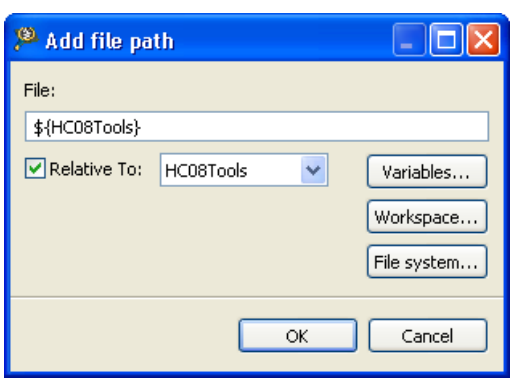

Figure 3-24. Tool Settings - HCS08 Compiler > Input - Add File Path Dialog Box

The following figure shows **Edit file path** dialog box.

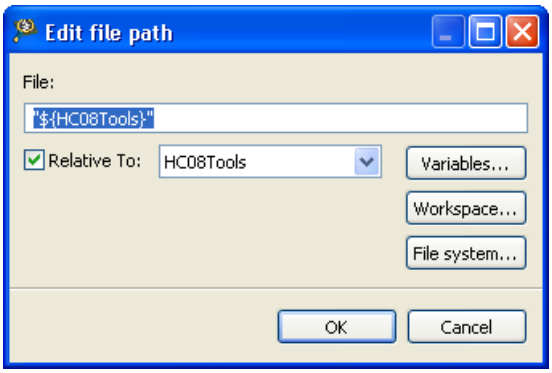

Figure 3-25. Tool Settings - HCS08 Compiler > Input - Edit File Path Dialog Box

The buttons in the **Add file path** and **Edit file path** dialog boxes help work with the paths.

- **OK-** Click to confirm the action and exit the dialog box.
- **Cancel-** Click to cancel the action and exit the dialog box.
- **Workspace -** Click to display the **File selection** dialog box and specify the path. The resulting path, relative to the workspace, appears in the appropriate list.
- **File system -** Click to display the **Open** dialog box and specify the path. The resulting path appears in the appropriate list.
- **Variables -** Click to display the **Select build variable** dialog box. The resulting variable appears in the path list.

3.8.2.6 HCS08 Compiler > Language

Use this panel to specify code-generation and symbol-generation options for HC(S)08 Compiler.

The following figure shows the **Language** panel.

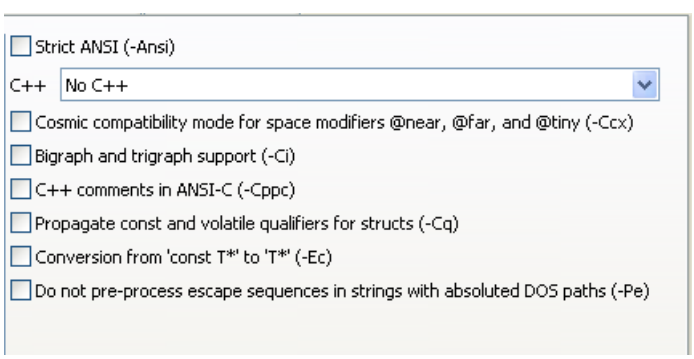

Figure 3-26. Tool Settings - HCS08 Compiler > Language

The following table lists the **Language** panel options for HC(S)08 Compiler.

Table 3-9. Tool Settings - HCS08 Compiler > Language Options

Option	Description
Strict `ANSI` (`-Ansi`)	Refer to the -Ansi: Strict ANSI topic.
C++	Refer to the -C++ (-C++f, -C++e, -C++c): C++ Support topic.
Cosmic compatibility mode for space modifiers `@near`, `@far`, and `@tiny` (`-Ccx`)	Refer to the -Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers topic.
Bigraph and trigraph support (`-Ci`)	Refer to the -Ci: Bigraph and Trigraph Support topic.
C++ comments in `ANSI-C` (`-Cppc`)	Refer to the -Cppc: C++ Comments in ANSI-C topic.
Propagate `const` and `volatile` qualifiers for `structs` (`-Cq`)	Refer to the -Cq: Propagate const and volatile Qualifiers for Structs topic.
Conversion from ` const T*' to ` T*' (`-Ec`)	Refer to the -Ec: Conversion from 'const T*' to 'T*' topic.
Do not pre-process escape sequences in strings with absoluted `DOS` paths (`-Pe`)	Refer to the -Pe: Do Not Preprocess Escape Sequences in Strings with Absolute DOS Paths topic.

3.8.2.7 HCS08 Compiler > Language > CompactC++ features

Use this panel to specify the options to disable the CompactC++ features for the HC(S)08 Compiler. The following figure shows the **CompactC++ features** panel.

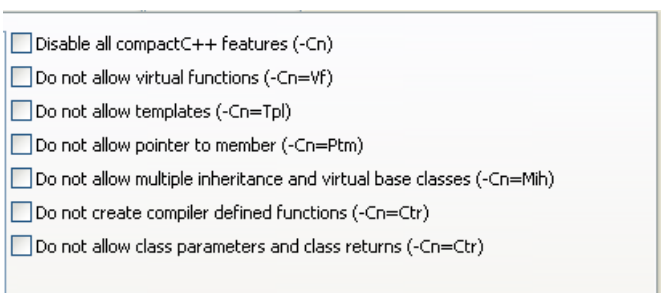

Figure 3-27. Tool Settings - HCS08 Compiler > Language > CompactC++ Features

NOTE

For information about the description of the options appearing in the **CompactC++ features** panel for HC(S)08 Compiler, refer to the

3.8.2.8 HCS08 Compiler > Host

Use this panel to specify the host options for HC(S)08 Compiler. The following figure shows the **Host** panel.

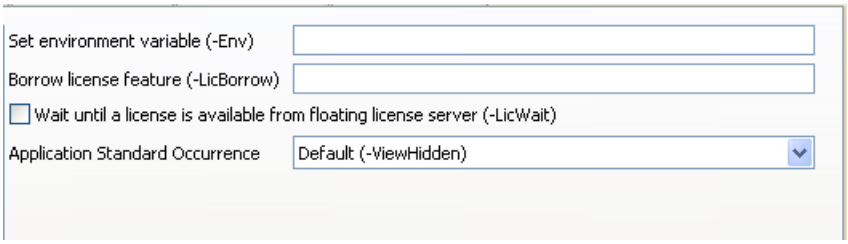

Figure 3-28. Tool Settings - HCS08 Compiler > Host

The following table lists and describes the options for **Host** panel.

Table 3-10. Tool Settings - HCS08 Compiler > Host Options

Option	Description
Set environment variable (`-Env`)	Refer to the -Env: Set Environment Variable topic.
Borrow license feature (`-LicBorrow`)	Refer to the -LicBorrow: Borrow License Feature topic.
Wait until a license is available from floating license server (`-LicWait`)	Refer to the -LicWait: Wait until Floating License is Available from Floating Li- cense Server topic.
Application Standard Occurrence	Refer to the -View: Application Standard Occurrence topic.

3.8.2.9 HCS08 Compiler > Code Generation

Use this panel to specify code-generation and symbol-generation options for the HC(S)08 Compiler. The following figure shows the **Code Generation** panel.

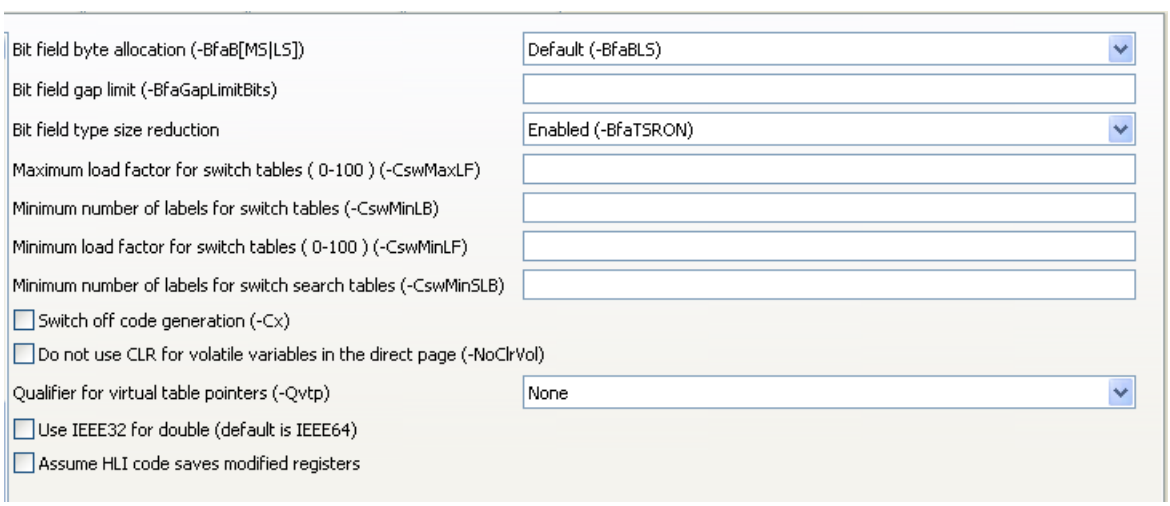

Figure 3-29. Tool Settings - HCS08 Compiler > Code Generation

The following table lists the **Code Generation** panel options for HC(S)08 compiler.

Table 3-11. Tool Settings - HCS08 Compiler > Code Generation Options

Option	Description
Bit field byte allocation (`-BfaB[MS|LS]`)	Refer to the -BfaB: Bitfield Byte Allocation topic.
Bit field gap limit (`-BfaGapLimitBits`)	Refer to the -BfaGapLimitBits: Bitfield Gap Limit topic.
Bit field type size reduction	Refer to the -BfaTSR: Bitfield Type Size Reduction topic.
Maximum load factor for switch tables (`0-100`) (`-CswMaxLF`)	Refer to the -CswMaxLF: Maximum Load Factor for Switch Tables topic.
Minimum number of labels for switch tables (`-CswMinLB`)	Refer to the -CswMinLB: Minimum Number of Labels for Switch Tables topic.
Minimum load factor for switch tables (`0-100`) (`-CswMinLF`)	Refer to the -CswMinLF: Minimum Load Factor for Switch Tables topic.
Minimum number of labels for switch search tables (`-CswMinSLB`)	Refer to the -CswMinSLB: Minimum Number of Labels for Switch Search Tables topic.
Switch off code generation (`-Cx`)	Refer to the -Cx: Switch Off Code Generation topic.
Do not use CLR for volatile variables in the direct page (`-NoClrVol`)	Refer to the -NoClrVol: Do not use CLR for volatile variables in the direct page topic.
Qualifier for virtual table pointers (`-Qvtp`)	Refer to the -Qvtp: Qualifier for Virtual Table Pointers topic.
Use `IEEE32` for double (default is `IEEE64`)	Refer to the -Fd: Double is IEEE32 topic.
Assume HLI code saves modified registers	Refer to the -Asr: It is Assumed that HLI Code Saves Written Registers topic.

3.8.2.10 HCS08 Compiler > Messages

Use this panel to specify the messages options for the HC(S)08 Compiler. The following figure shows the **Messages** panel.

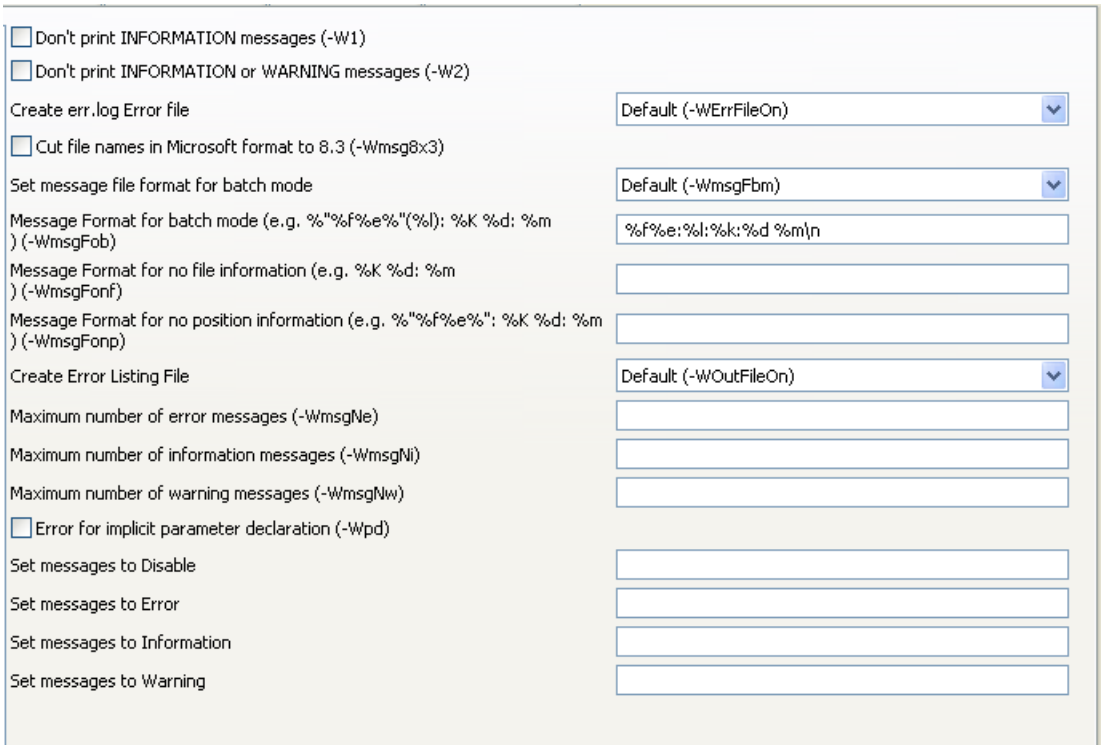

Figure 3-30. Tool Settings - HCS08 Compiler > Messages

The following table lists and describes the options for **Messages** panel.

Table 3-12. Tool Settings - HCS08 Compiler > Messages Options

Option	Description
Don't print `INFORMATION` messages (`-W1`)	Refer to the -W1: Don't Print Information Messages topic.
Don't print `INFORMATION` or `WARNING` messages (`-W2`)	Refer to the -W2: Do not Print INFORMATION or WARNING Messages topic.
Create `err.log` Error file	Refer to the -WErrFile: Create "err.log" Error File topic.
Cut file names in Microsoft format to 8.3 (`-Wmsg8x3`)	Refer to the -Wmsg8x3: Cut Filenames in Microsoft Format to 8.3 topic.
Set message file format for batch mode	Refer to the -WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode topic.
Message Format for batch mode (e.g. %"%f%e%"(%l): %K %d: %m) (`-WmsgFob`)	Refer to the -WmsgFob: Message Format for Batch Mode topic.
Message Format for no file information (e.g. %K %d: %m) (`-WmsgFonf`)	Refer to the -WmsgFonf: Message Format for No File Information topic.
Message Format for no position information (e.g. %"%f%e%": %K %d: %m) (`-WmsgFonp`)	Refer to the -WmsgFonp: Message Format for No Position Information topic.
Create Error Listing File	Refer to the -WOutFile: Create Error Listing File topic.
Maximum number of error messages (`-WmsgNe`)	Refer to the -WmsgNe: Maximum Number of Error Messages (enter <number>) topic.
Maximum number of information messages (`-WmsgNi`)	Refer to the -WmsgNi: Maximum Number of Information Messages (enter <number>) topic.

Table continues on the next page...

Table 3-12. Tool Settings - HCS08 Compiler > Messages Options (continued)

Option	Description
Maximum number of warning messages (`-WmsgNw`)	Refer to the -WmsgNw: Maximum Number of Warning Messages (enter <number>) topic.
Error for implicit parameter declaration (`-Wpd`)	Refer to the -Wpd: Error for Implicit Parameter Declaration topic.
Set messages to Disable	Enter the messages that you want to disable.
Set messages to Error	Enter the messages that you want to set as error.
Set messages to Information	Enter the messages that you want to set as information.
Set messages to Warning	Enter the messages that you want to set as warning.

3.8.2.11 HCS08 Compiler > Messages > Disable user messages

Use this panel to specify the options for disabling the user messages for the HC(S)08 Compiler.

The following figure shows the **Disable user messages** panel.

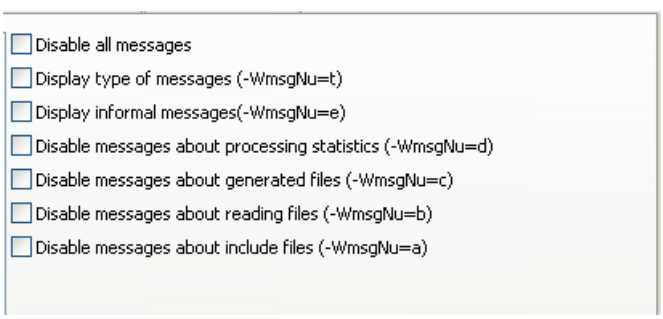

Figure 3-31. Tool Settings - HCS08 Compiler > Messages > Disable user messages

NOTE

For information about the description of the options available in the **Disable user messages** panel of HC(S)08 Compiler, refer to the -WmsgNu: Disable User Messages topic.

3.8.2.12 HCS08 Compiler > Preprocessor

Use this panel to specify preprocessor behavior. You can specify the file paths and define macros.

The following figure shows the **Preprocessor** panel.

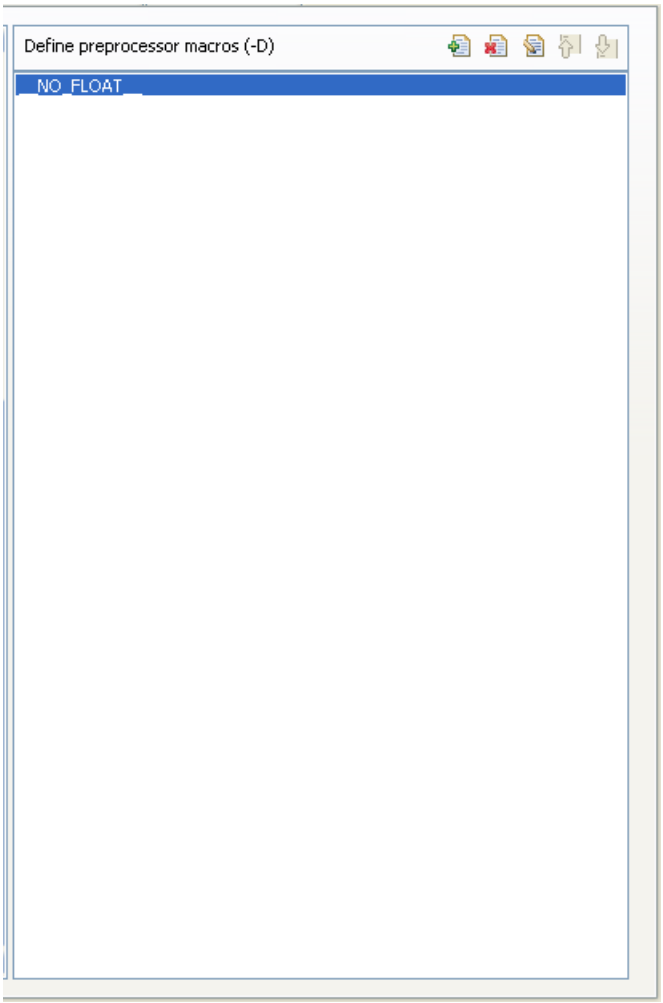

Figure 3-32. Tool Settings - HCS08 Compiler > Preprocessor

The following table lists the **Preprocessor** panel options for HC(S)08 Compiler.

Table 3-13. Tool Settings - HCS08 Compiler > Preprocessor Options

Option	Description
Define preprocessor macros (`-D`)	Defines, deletes, or rearranges the preprocessor macros. You can specify multiple macros and change the order in which the IDE uses the macros. Defines the preprocessor macros and optionally assign their values. This setting is equivalent to specifying the `-D name[=value]` command-line option. To assign a value, use the equal sign (=) with no white space. For example, this syntax defines a preprocessor value named `EXTENDED_FEATURE` and assigns `ON` as its value: `EXTENDED_FEATURE=ON`**Note:** If you do not assign a value to the macro, the shell assigns a default value of 1.

The following table lists and describes the toolbar buttons that help work with preprocessor macro definitions.

Table 3-14. Define Preprocessor Macros Toolbar Buttons

Button	Description
	**Add** - Click to open the **Enter Value** dialog box and specify the path/macro.
	**Delete** - Click to delete the selected path/macro.
	**Edit** - Click to open the **Edit Dialog** dialog box and update the selected path/macro.
	**Move up** - Click to move the selected path/macro one position higher in the list.
	**Move down** - Click to move the selected path/macro one position lower in the list.

The following figure shows the **Enter Value** dialog box.

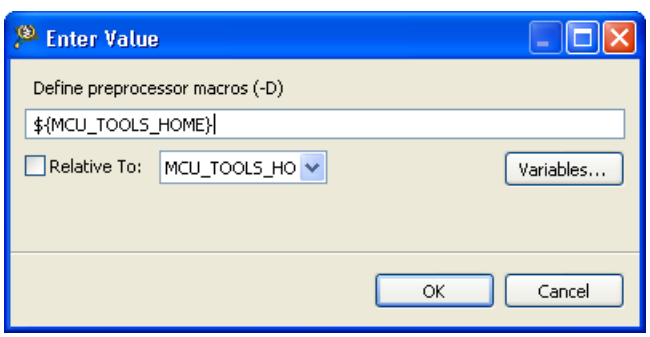

Figure 3-33. Tool Settings - HCS08 Compiler > Preprocessor - Enter Value Dialog Box

The following figure shows the **Edit Dialog** dialog box.

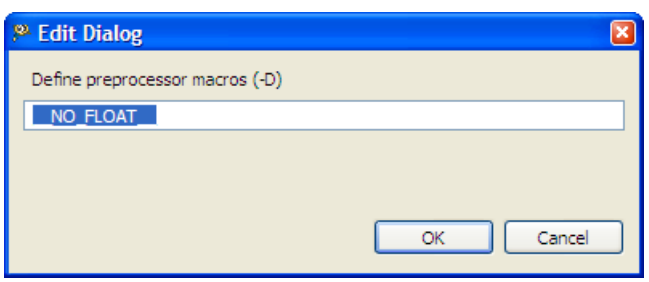

Figure 3-34. Tool Settings - HCS08 Compiler > Preprocessor - Edit Dialog Box

The buttons in the **Enter Value** and **Edit Dialog** dialog boxes help work with the preprocessor macros.

- **OK-** Click to *confirm* the action and exit the dialog box.
- **Cancel-** Click to *cancel* the action and exit the dialog box.
- **Variables-** Click to *select* the build variable.

3.8.2.13 HCS08 Compiler > Type Sizes

Use this panel to specify the type sizes for HC(S)08 Compiler. The following figure shows the **Type Sizes** panel.

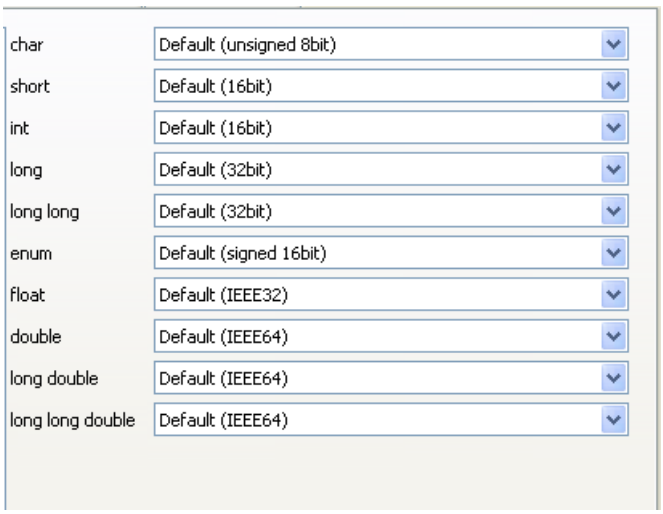

Figure 3-35. Tool Settings - HCS08 Compiler > Type Sizes

NOTE

For information about the description of the options appearing in the **Type Sizes** panel for HC(S)08 Compiler, refer to the -T: Flexible Type Management topic.

3.8.2.14 HCS08 Compiler > General

Use this panel to specify other flags for the HC(S)08 Compiler to use.

The following figure shows the **General** panel.

Figure 3-36. Tool Settings - HCS08 Compiler > General

The following table lists the **General** panel options for HC(S)08 compiler.

Table 3-15. Tool Settings - HCS08 Compiler > General Options

Option	Description
Other flags	Specifies the additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.

3.8.2.15 HCS08 Compiler > Optimization

Use this panel to control compiler optimizations. The compiler's optimizer can apply any of its optimizations in either global or non-global optimization mode. You can apply global optimization at the end of the development cycle, after compiling and optimizing all source files individually or in groups.

The following figure shows the **Optimization** panel.

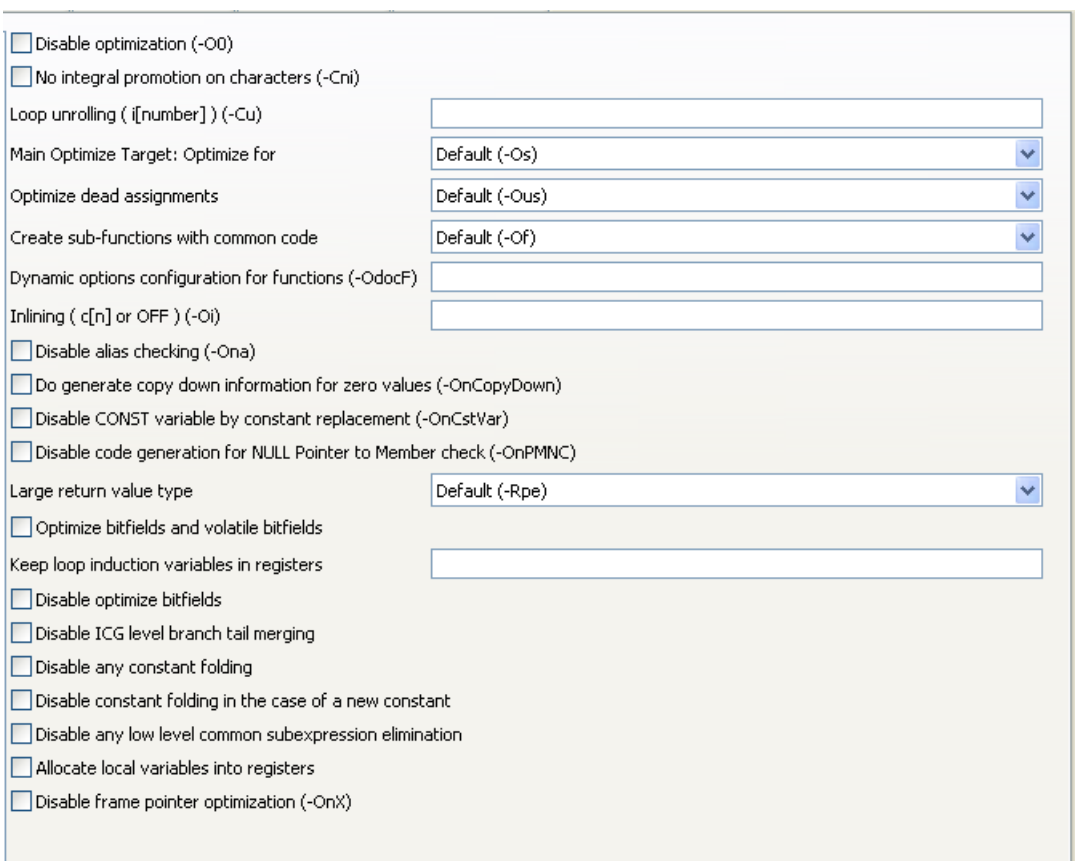

Figure 3-37. Tool Settings - HCS08 Compiler > Optimization

The following table lists the **Optimization** panel options for HC(S)08 compiler.

Table 3-16. Tool Settings - HCS08 Compiler > Optimization Options

Option	Description
Disable optimization (`-O0`)	Refer to the -O0 : Disable Optimizations topic.
No integral promotion on characters (`-Cni`)	Refer to the -Cni: No Integral Promotion on Characters topic.
Loop unrolling (`i[number]`) (`-Cu`)	Refer to the -Cu: Loop Unrolling topic.
Main Optimize Target: Optimize for	Refer to the -O(-Os, -Ot): Main Optimization Target topic.
Optimize dead assignments	Refer to the -Ous, -Ou, and -Onu: Optimize Dead Assignments topic.
Create sub-functions with common code	Refer to the -Of and-Onf: Create Sub-Functions with Common Code topic.
Dynamic options configuration for functions (`-OdocF`)	Refer to the -OdocF: Dynamic Option Configuration for Functions topic.
Inlining (`C[n] or OFF`) (`-Oi`)	Refer to the -Oi: Inlining topic.
Disable alias checking (`-Ona`)	Refer to the -Ona: Disable Alias Checking topic.
Do generate copy down information for zero values (`-OnCopyDown`)	Refer to the -OnCopyDown: Do Generate Copy Down Information for Zero Values topic.
Disable `CONST` variable by constant replacement (`-OnCstVar`)	Refer to the -OnCstVar: Disable CONST Variable by Constant Replacement topic.
Disable code generation for NULL Pointer to Member check (`-OnPMNC`)	Refer to the -OnPMNC: Disable Code Generation for NULL Pointer to Member Check topic.
Large return value type	Refer to the -Rp (-Rpe, -Rpt): Large Return Value Type topic.
Optimize bitfields and volatile bitfields	Refer to the -Obfv: Optimize Bitfields and Volatile Bitfields topic.
Keep loop induction variables in registers	Refer to the -Ol: Try to Keep Loop Induction Variables in Registers topic.
Disable optimize bitfields	Refer to the -Onbf: Disable Optimize Bitfields topic.
Disable ICG level branch tail merging	Refer to the -Onbt: Disable ICG Level Branch Tail Merging topic.
Disable any constant folding	Refer to the -Onca: Disable any Constant Folding topic.
Disable constant folding in the case of a new constant	Refer to the -Oncn: Disable Constant Folding in Case of a New Constant topic.
Disable any low level common subexpression elimination	Refer to the -One: Disable any Low Level Common Subexpression Elimination topic.
Allocate local variables into registers	Refer to the -Or: Allocate Local Variables into Registers topic.
Disable frame pointer optimization (`-OnX`)	Refer to the -OnX: Disable Frame Pointer Optimization topic.

3.8.2.16 HCS08 Compiler > Optimization > Tree optimizer

Use this panel to configure the tree optimizer options for the HC(S)08 compiler optimization.

The following figure shows the **Tree optimizer** panel.

Figure 3-38. Tool Settings - HCS08 Compiler > Optimization > Tree optimizer

NOTE

For information about the description of the options appearing in the **Tree optimizer** panel for HC(S)08 Compiler, refer to the -Ont: Disable Tree Optimizer topic.

3.8.2.17 HCS08 Compiler > Optimization > Optimize Library Function

Use this panel to optimize the library functions for the HC(S)08 compiler.

The following figure shows the Optimize Library Function panel.

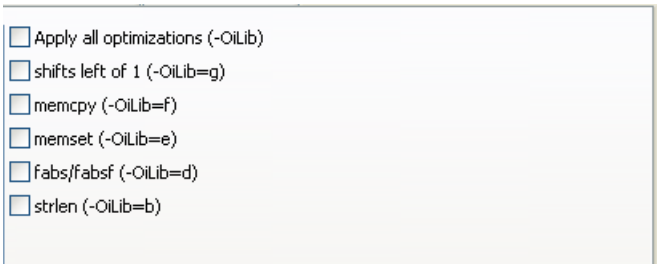

Figure 3-39. Tool Settings - HCS08 Compiler > Optimization > Optimize Library Function

NOTE

For information about the description of the options appearing in the **Optimize Library Function** panel for HC(S)08 compiler, refer to the -Oilib: Optimize Library Functions topic.

3.8.2.18 HCS08 Compiler > Optimization > Branch Optimizer

Use this panel to configure the branch optimizer for the HC(S)08 compiler. The following figure shows the **Branch Optimizer** panel.

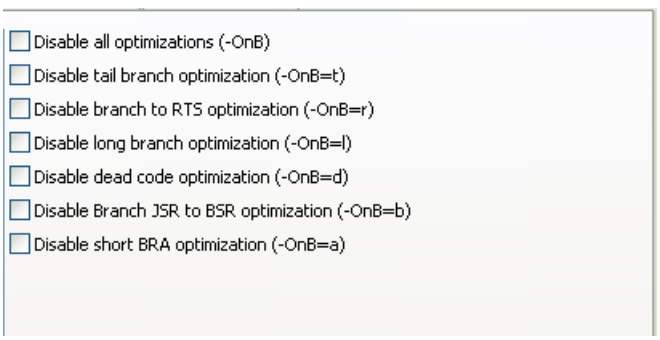

Figure 3-40. Tool Settings - HCS08 Compiler > Optimization > Branch Optimizer

NOTE

For information about the description of the options appearing in the **Branch Optimizer** panel for HC(S)08 compiler, refer to the -OnB: Disable Branch Optimizer topic.

3.8.2.19 HCS08 Compiler > Optimization > Peephole Optimization

Use this panel to configure peephole optimization for the HC(S)08 Compiler. The following figure shows the **Peephole Optimization** panel.

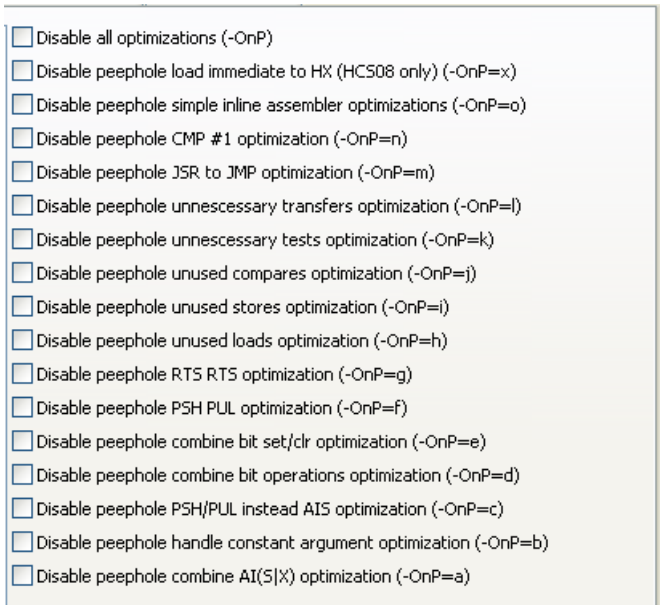

Figure 3-41. Tool Settings - HCS08 Compiler > Optimization > Peephole Optimization

NOTE

For information about the description of the options appearing in the **Peephole Optimization** panel for HC(S)08 compiler, refer to the -OnP: Disable Peephole Optimization topic.

3.8.3 CodeWarrior Tips and Tricks

You can use the following listed tips and tricks in the CodeWarrior for Microcontrollers V10.x:

- If Simulator or Debugger cannot be launched, check the *project's launch configuration settings*. For more information about the launch configurations and their settings, refer to the *Microcontrollers V10.x Targeting Manual*.

  NOTE

  The project's launch configurations can be viewed and modified from the IDE's **Run Configurations** or **Debug Configurations** dialog boxes. To open these dialog boxes, select **Run > Run Configurations** or **Run > Debug Configurations**.

- If a file cannot be added to the project, its file extension may not be available in the **File Types** panel. To resolve the issue, add the file's extension to the list in the **File Types** panel. To access the **File Types** panel, proceed as follows:

a. Right-click the desired project and select **Properties**.

   The **Properties for <project>** dialog box appears.

b. Expand the **C/C++ General** tree control and select **File Types**.
c. Select the **Use project settings** option.
d. Click **New**.

   The **C/C++ File Type** dialog box appears.

e. Enter the required pattern and select the file type and click **OK**.

   The **C/C++ File Type** dialog box closes.

f. Click **OK** to save the changes and close the properties window.

Now, you can add the required file to the project.

3.9 Integration into Microsoft Visual C++ 2008 Express Edition (Version 9.0 or later)

Use the following procedure to integrate the CodeWarrior tools into the Microsoft Visual Studio (Visual C++) IDE.

3.9.1 Integration as External Tools

1. Start Visual C++ 2008 Expression Edition.
2. Select **Tools > External Tools**.

   The **External Tools** dialog box appears.

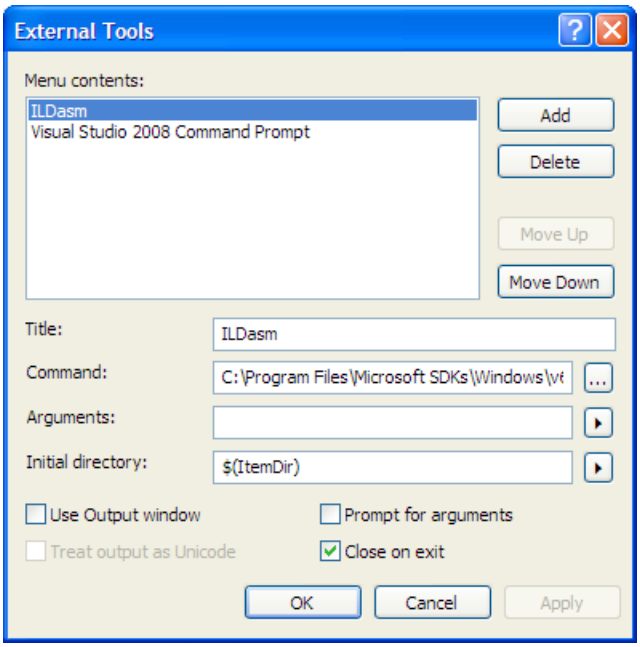

Figure 3-42. External Tools Dialog Box

3. Add a new tool by clicking **Add**.
4. In the **Title** text box, type the name of the tool to display in the menu, for example, *HCS08 Compiler*.
5. In the **Command** text box, either type the path of the `piper` tool or browse to `Piper.exe` using the **Browse** button. The location of `Piper.exe` is `<CWInstallDir>\MCU\prog\piper.exe`

   where `CWInstallDir` is the directory in which the CodeWarrior software is installed. `Piper.exe` redirects I/O so that the HC(S)08 build tools can be operated from within the Visual Studio UI.

6. In the **Arguments** text box, type the name of the tool to be launched. You can also specify any command line options, for example, `-Cc`, along with the `$(ItemPath)Visual` variable.

   You can use the pop-up menu to the right of the text box to enter other Visual variables. The text box should look like this:

   ```
   <CWInstallDir>\MCU\prog\chc08.exe -Cc $(ItemPath)
   ```

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

7. In the **Initial Directory** text box, use the pop-up menu to choose `$(ItemDir)`.
8. Check the **Use Output Window** checkbox.
9. Confirm that the **Prompt for arguments** checkbox is clear.
10. Click **Apply** to save your changes, then close the **External Tools** dialog box. The new tool appears in the **Tools** menu.

Similarly, you can display other build tools in the **Tools** menu. Use Build Tools (Application Programs) to obtain the file names of the other build tools and specify the file names in the **Arguments** text box.

This arrangement allows the active (selected) file to be compiled or linked in the Visual Editor. Tool messages are reported in the Visual **Output** window. You can double-click an error message in this window and the offending source code statement is displayed in the editor.

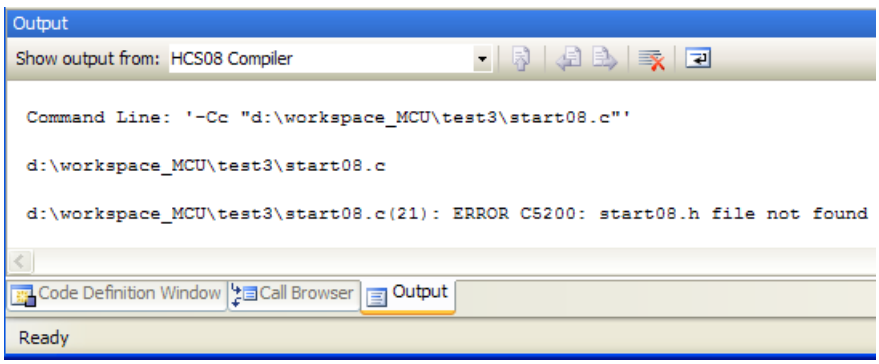

Figure 3-43. Visual Output Window

3.9.2 Integration with Visual Studio Toolbar

1. Start Visual C++ 2008 Expression Edition.
2. Make sure that all tools are integrated and appear as menu choices in the **Tools** menu.
3. Select **Tools > Customize** .

   The **Customize** window appears.

4. Select the **Toolbars** tab.
5. Select **New** and enter a name. For example, `Freescale Tools`.

   A new empty toolbar named Freescale Tools appears on your screen, adjacent to the **Customize** window.

6. Select the **Commands** tab.
7. From the **Category** drop-down list box, select **Tools** .
8. The right side of the window displays a list of commands. Search for the commands labeled `External Command x`, where `x` represents a number. This number corresponds to an entry in the **Menu contents** text box of the **External Tools** window. Count down the list in the **External Tools** window until you reach the first external tool you defined. For this example, it is **HCS08 Compiler** , and was the third choice on the

menu. Therefore, the desired tool command would be `External Command 3`. Alternately, with **Tools** selected in the **Customize** window's **Commands** tab, click on the **Tools** menu and it display the menu choices, with the external commands displayed.

9. Drag the desired command to the Freescale Tools toolbar.

   A button labeled **External Tool 3** appears on the **Freescale Tools** toolbar.

   Tip

   If the button appears dimmed, you have chosen an inactive external tool entry. Check your count value and try another external command entry.

10. Continue with this same sequence of steps to add the HCS08 Linker and HCS08 Assembler.
11. All of the default command names, such as **External Command 3** , **External Command 4** , and **External Command 5** on the toolbar are undescriptive, making it difficult to know which tool to launch. You must associate a name with them.
12. Right-click on a button in the Freescale Toolbar to open the button's context menu.
13. Select **Name** from the context menu.
14. Enter a descriptive name into the text field.
15. Repeat this process for all of the tools that you want to display in the toolbar.
16. Click **Close** to close the **Customize** window.

This enables you to start the CodeWarrior tools from the toolbar, as shown in the following figure.

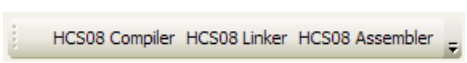

Figure 3-44. CodeWarrior Tools in Custom Visual Studio Toolbar

3.10 Compiler Modes and Language Extensions

It is possible to have the compiler operate in one of several modes to enforce certain programming language practices or to support special programming language extensions. You can:

- Force the compiler into a strict ANSI-C compliant mode, or
- Use language extensions that are specially designed for more efficient embedded systems programming.

3.10.1 C++, EC++, compactC++

The compiler supports the C++ language, if the C++ feature is enabled with a license file.

Some features of the C++ language are not designed for embedded controllers. If they are used, they may produce large amounts of code and require a lot of processor overhead, thereby sapping performance.

Avoid this situation by using the compactC++ and EC++ language extensions, which are subsets of the C++ language. Each subset is adapted for embedded application programming.

These subsets of the C++ language avoids implicit and explicit overhead of the C++ language (for example, virtual member functions and multiple inheritance). The EC++ is a restricted subset, while compactC++ includes features which are not in the EC++ definition. This makes it more flexible.

Another key aspect of compactC++ is its flexible configuration of the language (for example, allowed keywords, code generation behavior, and message management). The compiler is adapted for the special needs for embedded programming.

The compiler provides the following language settings:

- ANSI-C: The compiler behaves as an ANSI-C compiler. It is possible to force the compiler into a strict ANSI-C compliant mode, or to use language extensions designed for efficient embedded systems programming.
- EC++: The compiler behaves as a C++ compiler. The following features are not allowed in EC++:
  - Mutable specifier
  - Exception handling
  - Runtime type identification
  - Namespace
  - Template
  - Multiple inheritance
  - Virtual inheritance
  - Library support for `w_char` and `long double`
- cC++, compactC++: In this mode, the compiler behaves as a full C++ compiler that allows the C++ language to be configured to provide compact code. This enables developers to enable/disable and configure the following C++ features:
  - Multiple inheritance
  - Virtual inheritance
  - Templates
  - Trigraph and bigraph

- Compact means:
  - No mutable qualifier
  - No exception handling
  - No runtime type identification
  - No namespaces
  - No library support for `w_char` and `long double`
- C++: The compiler behaves as a full C++ compiler. However, because the C++ language provides some features not usable for embedded systems programming, such features may be not usable.

NOTE

Currently, for the MCU V10.x release, the Eclipse UI only allows you to turn C++ language support on or off. The UI currently does not implement settings for EC++ and compactC++ language extensions. You must use command line arguments to specify the compiler's support for these language extensions. More details on these arguments can be found in Compiler Options chapter, under -C++ (-C++f, -C++e, -C++c): C++ Support.

3.11 Object-File Formats

The compiler supports two different object-file formats: ELF/DWARF and the vendor-specific HIWARE object-file format. The object-file format specifies the format of the object files (`*.o`), library files (`*.lib`), and the absolute files (`*.abs`).

NOTE

Do not mix object-file formats. Both the HIWARE and the ELF/DWARF object files use the same filename extensions.

3.11.1 HIWARE Object-File Format

The HIWARE Object-File Format is a vendor-specific object-file format defined by HIWARE AG. This compact object-file format produces smaller file sizes than the ELF/DWARF object files. This enables faster file operations. Third-party tool vendors find this compact file size easy to support (for example, emulators from Abatron, Lauterbach, or iSYSTEM). This object-file format supports both ANSI-C and Modula-2.

However, third-party tool vendors must support this object-file format explicitly. Note that there is a lack of extensibility, a lack of debug information, and no C++ support. For example, HIWARE Object-file Format does not support the full flexibility of Compiler Type Management.

Using the HIWARE object-file format may also result in slower source or debug information loading. In the HIWARE object-file format, the source position information is provided as position information (offset in file), and not directly in a file, line, or column format. The debugger must translate this HIWARE object-file source information format into a file, line, or column format. This tends to slow down the source file or debugging information loading process.

3.11.2 ELF/DWARF Object-File Format

The ELF/DWARF object-file format originally comes from the UNIX world. This format is very flexible and supports extensions.

Many chip vendors define this object-file format as the standard for tool vendors supporting their devices. This standard allows inter-tool operability making it possible to use the compiler from one tool vendor, and the linker from another. The developer has the choice to select the best tools for the tool chain. In addition, other third parties (for example, emulator vendors) need only support this object file to support a wide range of tool vendors.

Object-file sizes are larger when compared with the HIWARE object-file format. This object-file format also supports ANSI-C.

3.11.3 Tools

The CodeWarrior Suite contains the following tools, among others:

- Compiler

  The same compiler executable supports both object-file formats. Use the -F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format compiler option to switch the object-file format.

  Note that not all compiler backends support both ELF/DWARF and the HIWARE object-file formats. Some only support one of the two.

- Decoder

  Use the same executable, `decoder.exe`, for both the HIWARE and the ELF/DWARF object-file formats.

- Linker

  Use the same executable, `linker.exe`, for both the HIWARE and the ELF/DWARF object-file formats.

- Simulator or Debugger

  The simulator and the debugger support both object-file formats.

3.11.4 Mixing Object-File Formats

You cannot mix HIWARE and ELF object files. You can mix ELF object files with DWARF 1.1 and DWARF 2.0 debug information. However, the final generated application contains no debug data.

Chapter 4
Graphical User Interface

Graphical User Interface (GUI) provides a simple yet powerful user interface for the CodeWarrior build tools. Following are the features of the CodeWarrior build tools GUI:

- An interactive mode for those who need an easy way to modify the tool settings
- A batch mode that allows the tools to interoperate with a command-line interface (CLI) and make files
- Online Help
- Error Feedback
- Easy integration into other tools, for example, CodeWarrior IDE, CodeWright, Microsoft Visual Studio, and WinEdit

This chapter describes the GUI and provides useful hints. Its major topics are:

4.1 Launching Compiler

You can use either of the ways to launch the HC(S)08 Compiler:

- Navigate to < `CWInstallDir>\MCU\prog` using Windows Explorer and double-click `chc08.exe`, where `CWInstallDir` is the directory in which the CodeWarrior software is installed.
- Create a shortcut for `chc08.exe` on the desktop and double-click the shortcut
- Create a shortcut for `chc08.exe` in the **Start > Programs** menu and select the shortcut
- Use batch and com mand files
- Use other to ols, such as Editor and Visual Studio

You can launch the compiler in either Interactive Mode or Batch Mode.

4.1.1 Interactive Mode

If you start the compiler with no options and no input files, the tool enters the interactive mode and the compiler GUI is displayed. This is usually the case if you start the compiler using the Windows Explorer or an Desktop or Programs menu icon.

The following code specifies an input line associated with a desktop icon for the HC(S)08 compiler:

```
chc08.exe -F2 a.c d.c
```

4.1.2 Batch Mode

If you are using the command prompt to start the compiler and specify arguments, such as options and/or input files, the compiler starts in the batch mode. In the batch mode, the compiler does not open a window or does not display GUI. The taskbar displays the progress of the compilation processes, such as processing the input.

The following code presents a command to run the HC(S)08 compiler and process two source files, `a.c` and `d.c`, which appear as arguments:

```
C:\Freescale\CW MCU v10.x\MCU\prog>chc08.exe - F2 a.c d.c
```

The compiler redirects message output, `stdout` using the redirection operator, ` >'. The redirection operation instructs the compiler to write the message output to a file. The following code redirects command-line message output to a file, `myoutput.o`:

```
C:\Freescale\CW MCU v10.x\MCU\prog>chc08.exe - F2 a.c d.c > myoutput.o
```

The command line process returns after the compiling process starts. It does not wait until the previous process has terminated. To start a tool and wait for termination , for example, to synchronize successive operations, use the `start` command in the Windows® 2000, Windows XP, or Windows Vista™ operating systems, or use the `/wait` options (refer to the Windows Help `` `help start' ``). Using `start /wait` pairs, you can write perfect batch files. The following code starts a compilation process but waits for the termination of the prior activities before beginning:

```
C:\Freescale\CW MCU v10.x\MCU\prog>start/wait chc08.exe -F2 a.c d.c
```

4.2 Compiler Main Window

If you do not specify a filename while starting the Compiler, the **CompilerMain Window** is empty on startup.

The Compiler window consists of a window title, a menu bar, a toolbar, a content area, and a status bar.

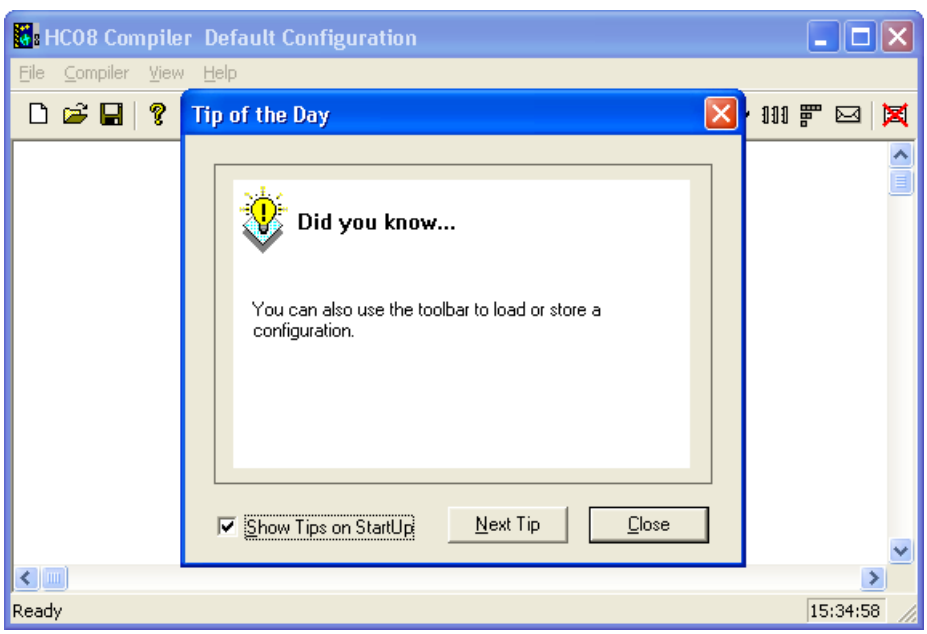

Figure 4-1. Compiler Main Window

When the Compiler starts, the **Tip of the Day** dialog box displaying the latest tips, appears in the Compiler main window.

- The **Next Tip** button displays the next tip about the Compiler.
- To disable the **Tip of the Day** dialog box from opening automatically when the application starts, clear the **Show Tips on StartUp** checkbox.

NOTE

This configuration entry is stored in the local project file.

- To enable automatic display of the **Tip of the Day** dialog box when the Compiler starts, select **Help > Tip of the Day** . The **Tip of the Day** dialog box opens. Check the **Show Tips on StartUp** checkbox.
- Click **Close** to close the **Tip of the Day** window.

The IDE enables/disables the Tip of the Day as per the option you selected.

This section covers the following topics:

4.2.1 Window Title

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

4.2.2 Content Area

The content area displays the logging information about the compilation process. This logging information consists of:

- the name of the file being compiled,
- the whole name, including full path specifications, of the files processed, such as C source code files and all of their included header files

- the list of errors, warnings, and information messages
- the size of the code generated during the compilation session

When you drag and drop a file into the Compiler window content area, the Compiler either loads it as a configuration file or compiles the file. It loads the file as configuration file, if the file has the `*.ini` extension. Otherwise, the compiler processes the file using the current option settings.

All of the text in the Compiler 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 source and include files, for output messages that concern a specific file, and for all output that is directed to a text file. If a file context is available, double-clicking on the text or message opens this file in an editor, as specified in the Editor Configuration. Also, a right-click in this area opens a context menu. The context menu contains an **Open** entry if a file context is available. If a context menu entry is present but the file cannot be opened, refer to the Global Initialization File (mcutools.ini).

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 `F1`.

  If the selected line does not have a message number, the main help displays.

- *Press* `Shift-F1` and then click on the message text.

  If the selected line does not have a message number, the main help displays.

- Right click the message text and select **Help on** .

  This entry is available only if a message number is available in the **HC08 Compiler Message Settings Dialog Box** .

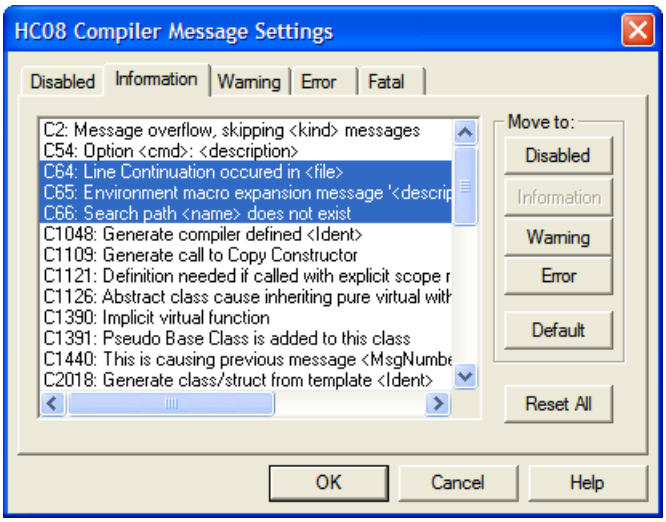

Figure 4-2. HC08 Compiler Message Settings Dialog Box

4.2.3 Toolbar

The toolbar consists of an array of buttons and a content box, just above the content area. Starting from left to right, these elements are:

- The three left-most buttons in the toolbar correspond to the **New** , **LoadConfiguration** , and **Save Configuration** entries of the **File** menu.
- The **Help** button opens on-line help window.
- The **Context Help** button, when clicked, has the cursor change its form and display a question mark beside the pointer. Clicking an item calls the help file. Click on menus, toolbar buttons, or the window area to get information specific to that item.

NOTE

You can also access context help by typing the key shortcut, `Shift-F1`.

Figure 4-3. Toolbar

- The command line history text box is in the middle of the toolbar and displays the command line history. It contains a list of the commands executed previously. Once you select a command or enter it in the history text box, clicking the **Compile** button to the right of the command line history executes the command. The `F2` keyboard shortcut key jumps directly to the command line. In addition, a context menu is associated with the command line.
- The **Stop** button stops the current process session.

- The next four buttons open the **Options** , **Smart Slider** , **Standard Types** , and **Messages** dialog boxes. These dialog boxes are used to modify the compiler's settings, such as language used, optimization levels, and size of various data types. The use of the dialog boxes are discussed later in the chapter.
- The right-most button clears the content area (Output Window).

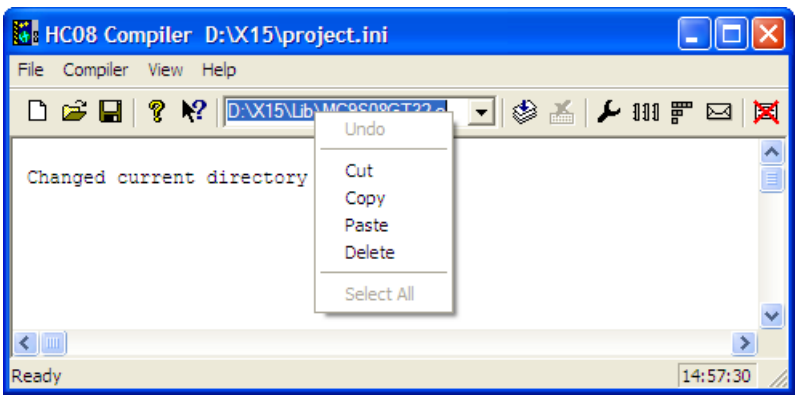

Figure 4-4. Command Line Context Menu

4.2.4 Status Bar

The status bar displays the current status of the compiler when it processes a file.When the compiler is idle, the message area displays a brief description of function of the button or menu entry when you place the cursor over an item on the toolbar. The following figure shows the status bar of the compiler.

Figure 4-5. Status Bar

4.2.5 Compiler Menu Bar

The following table lists the menus available in the Compiler menu bar.

Table 4-1. Menu Items in Menu Bar

Menu Item	Description
File Menu	Manage the Compiler's configuration files
Compiler Menu	Set the Compiler's options
View Menu	Customize the Compiler's window output
Help Menu	Access the standard Windows Help menu

Figure 4-6. Menu Bar

4.2.6 File Menu

The **File** menu is used to save or load Compiler configuration files. A Compiler configuration file contains the following information:

- Compiler option settings specified in the settings panels
- Message Settings that specify which messages to display and which messages to treat as error messages
- List of the last command lines executed and the current command line being executed
- Window position information
- Tip of the Day settings, including the current entry and enabled/disabled status

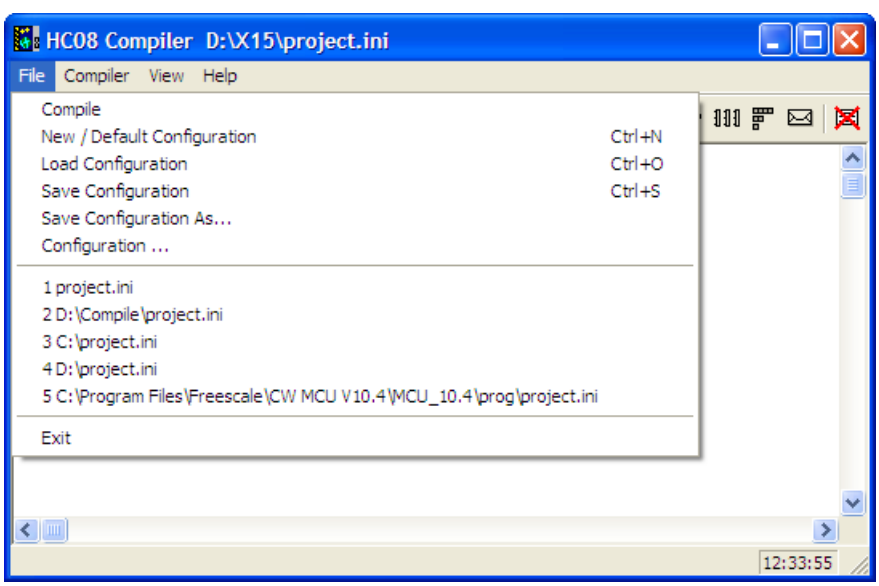

Figure 4-7. File Menu

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.

The following table lists the File menu options.

Table 4-2. File Menu Options

Menu Option	Description
Compile	Displays the **Select File to Compile** dialog box. Browse to the desired input file using the dialog box. Select the file and click **OK**. The Compiler compiles the selected file.
New/Default Configuration	Resets the Compiler option settings to their default values. The default Compiler options which are activated are specified in the Compiler Options chapter.
Load Configuration	Displays the **Loading configuration** dialog box. Select the desired project configuration using the dialog box and click **Open**. The configuration data stored in the selected file is loaded and used in further compilation sessions.
Save Configuration	Saves the current settings in the configuration file specified on the title bar.
Save Configuration As	Displays a standard **Save As** dialog box. Specify the configuration file in which you want to save the settings and click **OK**.
Configuration	Opens the **Configuration** dialog box to specify the editor used for error feedback and which parts to save with a configuration. Refer to the topics, Editor Settings Dialog Box and Save Configuration Dialog Box.
1 project.ini 2 D:\X15\project.ini	Recent project configuration files list. This list can be used to reopen a recently opened project configuration.
Exit	Closes the Compiler.

4.2.7 Compiler Menu

Use the **Compiler** menu to customize the Compiler. It allows you to set certain compiler options graphically, such as the optimization level. The table below lists the **Compiler** menu options.

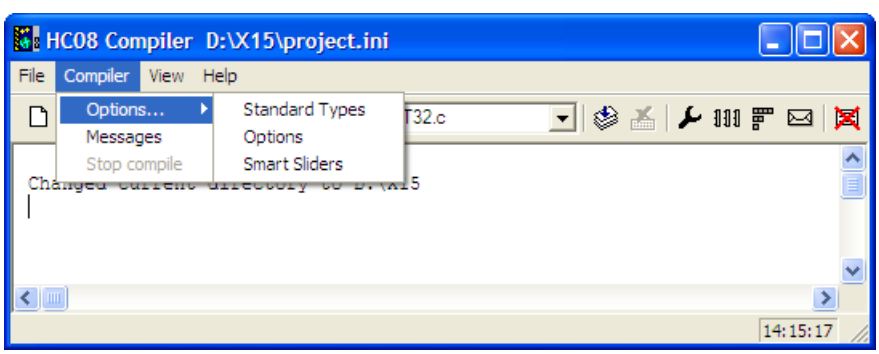

Figure 4-8. Compiler Menu

Table 4-3. Comp iler Menu Options

Menu Option	Description
Opt ions	Customizes the Compiler. You can graphically set or reset the compiler options.
Standard Types	Specifies the size you want to associate with each ANSI C standard type. (Refer to the topic, Standard Types Settings Dialog Box.)
Options	Defines the options that the compiler uses when processing an input file. (Refer to the topic, Option Settings Dialog Box.)
Smart Slider	Defines the optimization level you want the compiler to use when processing the input file. (Refer to the topic, Smart Control Dialog Box.)
Messages	Allows you to map the different error, warning, or information messages to another message class. (Refer Message Settings Dialog Box.)
Stop Compile	Immediately stops the current processing session.

4.2.8 View Menu

Use the **View** menu to customize the Compiler window, such as display or hide the status bar or toolbar, set the font used in the window, or clear the window's content area.The table below lists the **View Menu** options.

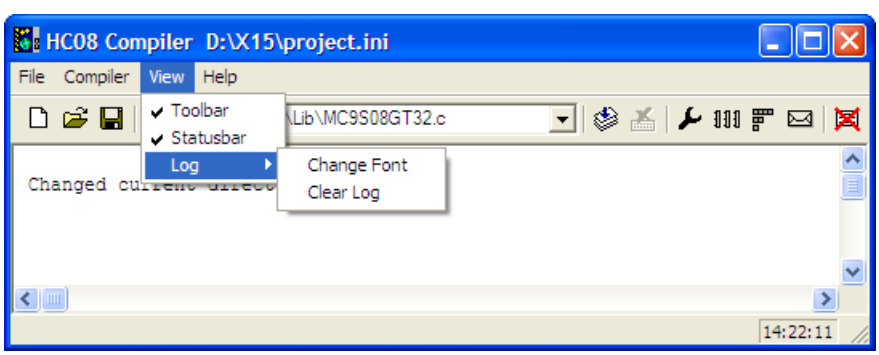

Figure 4-9. View Menu

Table 4-4. View Menu Options

Menu Option	Description
Toolbar	Hides or displays the toolbar in the Compiler window.
Status Bar	Hides or displays the status bar in the Compiler window.
Log	Customize the output in the Compiler's window content area. The following entries are available when **Log** is selected.

Table continues on the next page...

Table 4-4. View Menu Options (continued)

Menu Option	Description
Change Font	Opens a standard font selection box. Applies the selected options to the Compiler's window content area.
Clear Log	Clears the Compiler's window content area.

4.2.9 Help Menu

Use the **Help** menu to activate or deactivate the **Tip of the Day** dialog box at application startup. In addition, it provides access a standard Windows Help file and an **About** box. The table below lists the **Help** menu options.

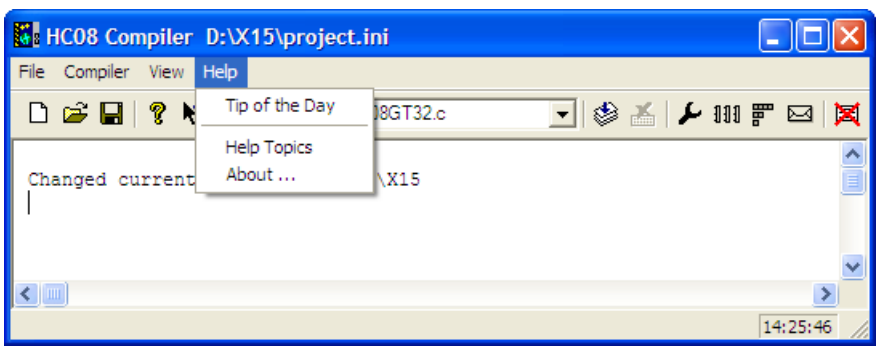

Figure 4-10. Help Menu

Table 4-5. Help Menu Options

Menu Option	Description
Tip of the Day	Activates or deactivates Tip of the Day at startup.
Help Topics	Standard Help topics.
About	Displays an About dialog box with version and licensing information.

4.3 Editor Settings Dialog Box

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

These are the main entries for the Editor configuration:

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

4.3.1 Global Editor (shared by all tools and projects)

All tools and projects on one work station share the **Global Editor** option. The `mcutools.ini` global initialization file stores the configuration in the `[Editor]` section. The editor command line specifies some Modifiers.

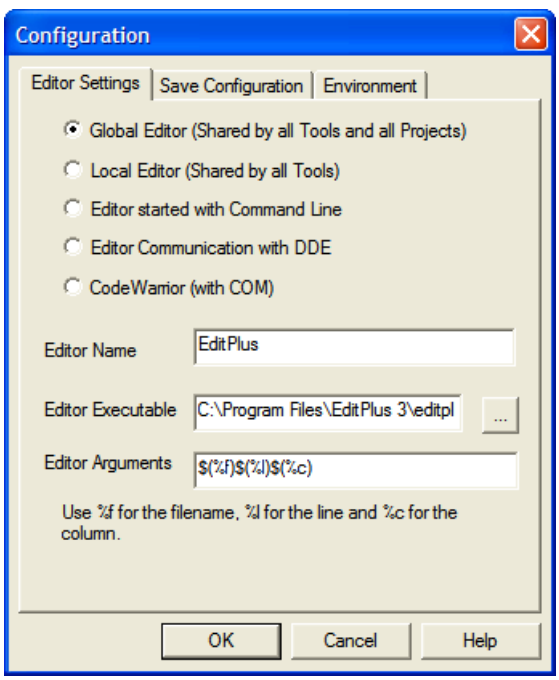

Figure 4-11. Global Editor Configuration Dialog Box

4.3.2 Local Editor (shared by all tools)

All tools using the same project file share the **Local Editor** option. Modifying an entry of the **Global** or **Local** configuration modifies the behavior of the other tools using the same entry when these tools are restarted.

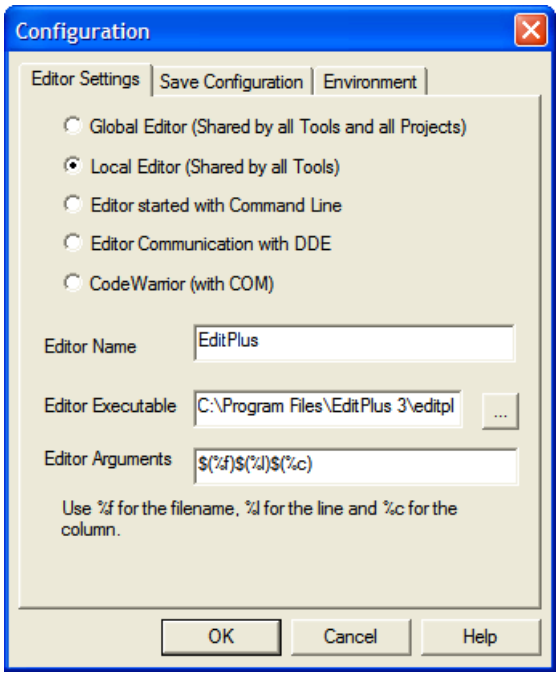

Figure 4-12. Local Editor Configuration Dialog Box

4.3.3 Editor Started with Command Line

Selecting this editor option, as shown in the figure below, associates a separate editor for error feedback with the application. The configured editor (that is, Global or Local) is not used for error feedback.

In the **Command Line** text box, enter the command that starts the editor.

The format of the editor command depends on the syntax. Specify Modifiers in the editor command line to refer to specific line numbers in the file. (Refer Modifiers.)

The format of the editor command depends upon the syntax that is used to start the editor.

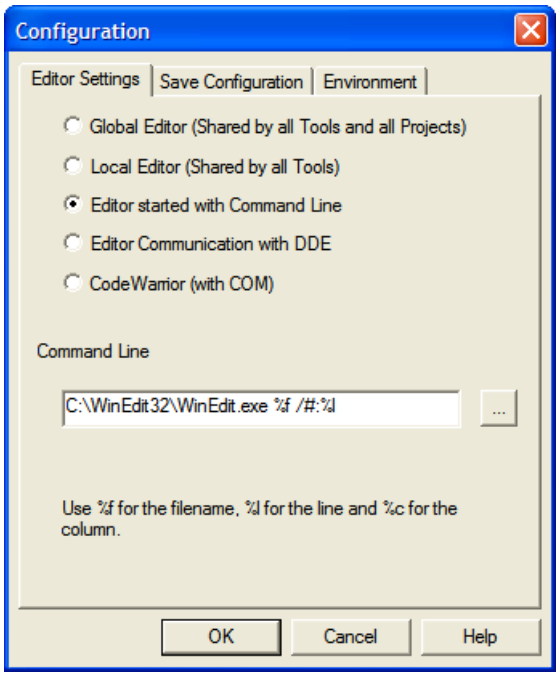

Figure 4-13. Editor Started with Command Line

4.3.3.1 Examples

For CodeWright V6 editor, use the following command line (with an adapted path to the `cw32.exe` file):

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

For the WinEdit 32-bit version, use the following command line (with an adapted path to the `winedit.exe` file):

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

4.3.4 Editor Started with DDE

Enter the **Service Name** , **Topic Name** , and the **Client Command** for the DDE connection to the editor. The **Topic Name** and **Client Command** entries can have modifiers for the filename, line number, and column number as explained in Modifiers.

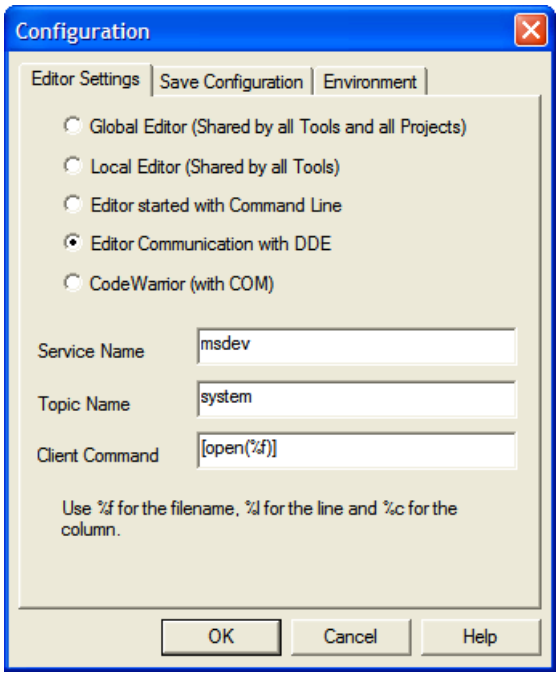

Figure 4-14. Editor Started with DDE (Microsoft Developer Studio)

For Microsoft Developer Studio, use the settings in the following listing.

Listing: Microsoft Developer Studio Configuration

```
Service Name    :
msdev

Topic Name      : system


Client Command  : [open(%f)]
```

UltraEdit-32 is a powerful shareware editor. It is available from www.idmcomp.com or www.ultraedit.com (email idm@idmcomp.com). For UltraEdit, use the settings in the following listing.

Listing: UltraEdit-32 Editor Settings

```
Service Name   : UEDIT32
Topic Name     : system

Client Command : [open("%f/%l/%c")]
```

NOTE

The DDE application (for example, Microsoft Developer Studio or UltraEdit) must be running, or DDE communications will fail.

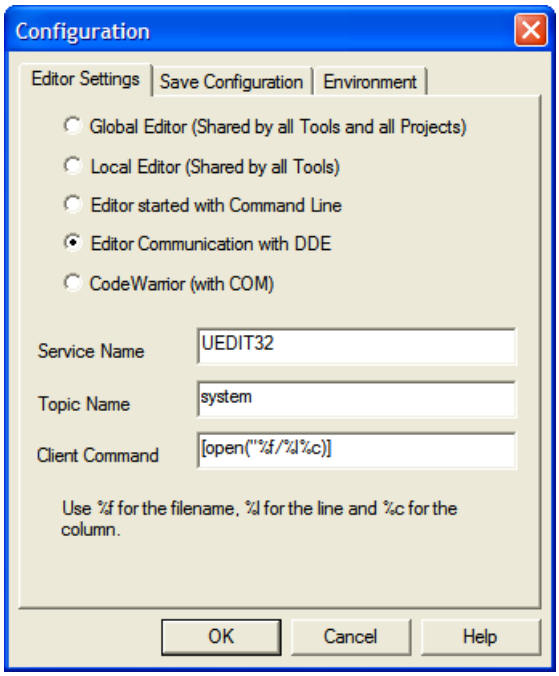

Figure 4-15. Editor Started with DDE (UltraEdit-32)

4.3.5 CodeWarrior (with COM)

Select **CodeWarrior (with COM)** to use the CodeWarrior IDE as the editor. This registers CodeWarrior IDE as the COM server by the installation script.

CAUTION

This panel supports use of the legacy CodeWarrior IDE. It is not for use with the Eclipse IDE that hosts the CodeWarrior for Microcontrollers V10.x.

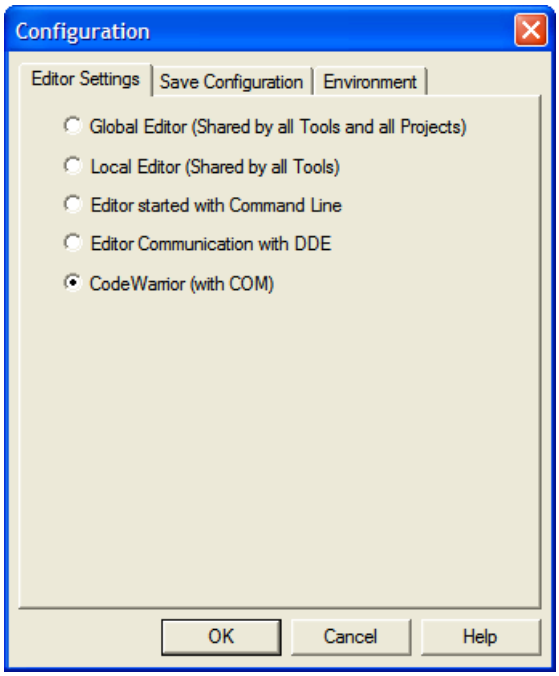

Figure 4-16. CodeWarrior (with COM)

4.3.6 Modifiers

The configuration must contain modifiers that instruct the editor which file to open and at which line.

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

NOTE

The `%l` modifier can only be used with an editor which is started with a line number as a parameter, except when working with WinEdit version 3.1 or lower or with Notepad. When working with these editors, start the editor 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 the editor manual for the command line definition for starting the editor.

4.4 Save Configuration Dialog Box

Select the **Save Configuration** tab to access the **Save Configuration** dialog box. Specify which portions of your configuration to save to the project file.

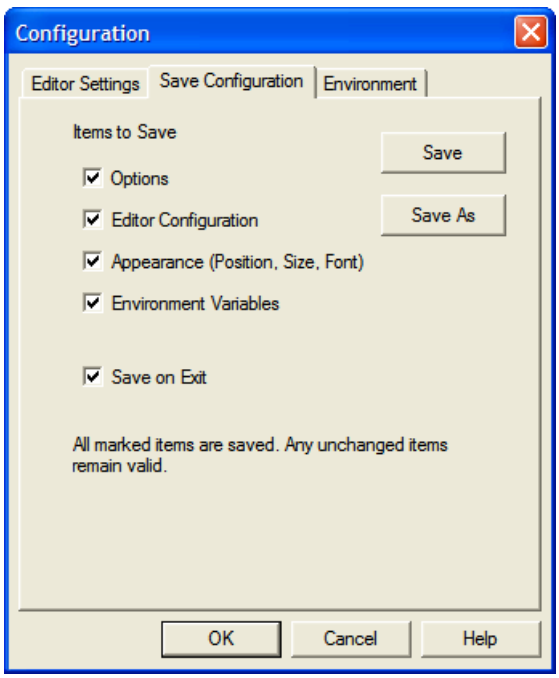

Figure 4-17. Save Configuration Dialog Box

The **Save Configuration** dialog box offers the following options:

- Options

  Check this option to save the current option and message settings when a configuration is written. By disabling this option, the last-saved content remains valid.

- Editor Configuration

  Check this option to save the current editor setting when a configuration is written. By disabling this option, the last-saved content remains valid.

- Appearance

  Check this option to save items such as the window position (only loaded at startup time) and the command line content and history. By disabling this option, the last-saved content remains valid.

- Environment Variables

  Check this option to save the environment variable changes made in the **Environment** configuration dialog box.

  NOTE

  By clearing specific option, only some parts of a configuration file are written. For example, clear the **Options** checkbox when the best options are found. Subsequent future save commands no longer modify these options.

- Save on Exit

  The application saves the current configuration on exit. No save confirmation dialog is displayed. If this option is not set, the application will not write the configuration at exit, even if parts of the configuration have changed. No save confirmation appears in either case when closing the application.

  Most settings are stored in the configuration file only. The only exceptions are:

  - The recently used configuration list.
  - All settings in this dialog box.

  NOTE

  The tool configurations can (and in fact are intended to) coexist in the same file as the project configuration of UltraEdit-32. When the shell configures an editor, the application reads this content out of the project file (`project.ini`), if present. This filename is also suggested (but not required) to be used by the tool.

4.5 Environment Configuration Dialog Box

Open the **Environment Configuration** dialog box to configure the environment. The dialog box's content is read from the `[ Environment Variables]` section of the actual project file.

The following listing shows the available environment variables.

Listing: Environment aariables

```
General Path:       GENPATH
Object Path:        OBJPATH
```

```
Text Path:          TEXTPATH
Absolute Path:      ABSPATH
Header File Path:   LIBPATH
Various Environment Variables: other variables not mentioned above.
```

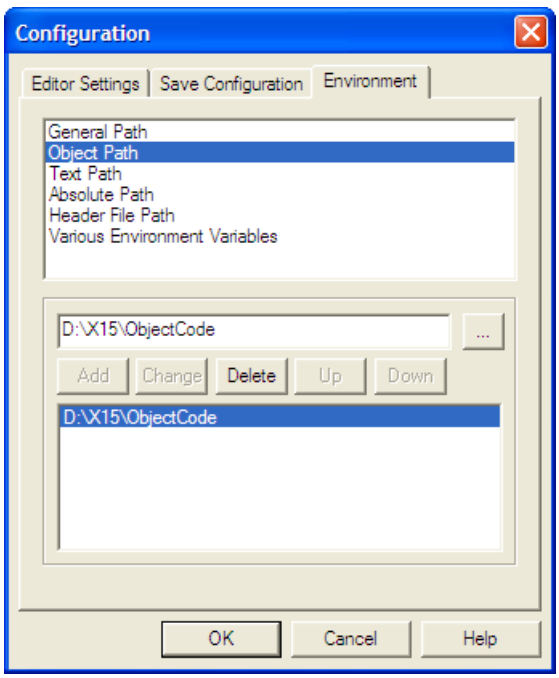

Figure 4-18. Environment Configuration Dialog Box

The following buttons are available in this dialog box.

Table 4-6. Environment Configuration Dialog Box Button Functions

Button	Function
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

The variables are written to the project file only if you save the configuration by either, select **File > Save Configuration** or press **CTRL+S**. In addition, you can specify in the **Save Configuration as** dialog box if the environment is written to the project file or not.

4.6 Standard Types Settings Dialog Box

Use the **Standard Types Settings** dialog box to define the size to associate with each ANSI C standard type. You can also use the -T: Flexible Type Management compiler option to configure ANSI-C standard type sizes.

NOTE

Not all formats may be available for a target. In addition, there must be at least one type for each size. For example, it is invalid to specify all types to a size of 32 bits. Also, the HIWARE Format does not support a size greater than 8 bits for the `char` type.

The following listing lists the rules that apply when you modify the size associated with an ANSI-C standard type:

Listing: Size Relationships for the ANSI-C Standard Types.

```
sizeof(char)   <= sizeof(short)
sizeof(short)  <= sizeof(int)

sizeof(int)    <= sizeof(long)

sizeof(long)   <= sizeof(long long)

sizeof(float)  <= sizeof(double)

sizeof(double)<= sizeof(long double)
```

Enumerations must be smaller than or equal to `int`.

Use the **signed** checkbox to specify whether the `char` or `enum` type must be considered as signed or unsigned for your application.

The **Defaults** button resets the size of the ANSI C standard types to their default values. The ANSI C standard type default values depend on the target processor.

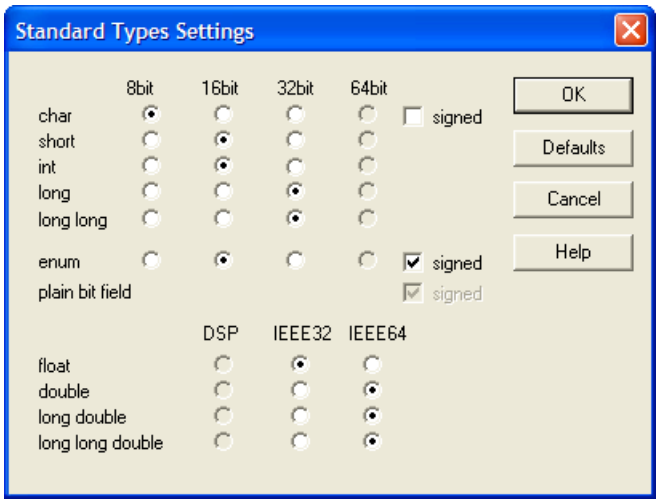

Figure 4-19. Standard Types Settings Dialog Box

4.7 Option Settings Dialog Box

Use the **Option Settings** dialog box) to set or reset tool options. This panel also displays the possible command line arguments used to invoke these options in the lowest pane. The available options are arranged into different groups, with separate tabs for each group. The content of the list box depends on the selected tab. The table below lists the Option Settings selections.

NOTE

Not all of the option tabs may be available.

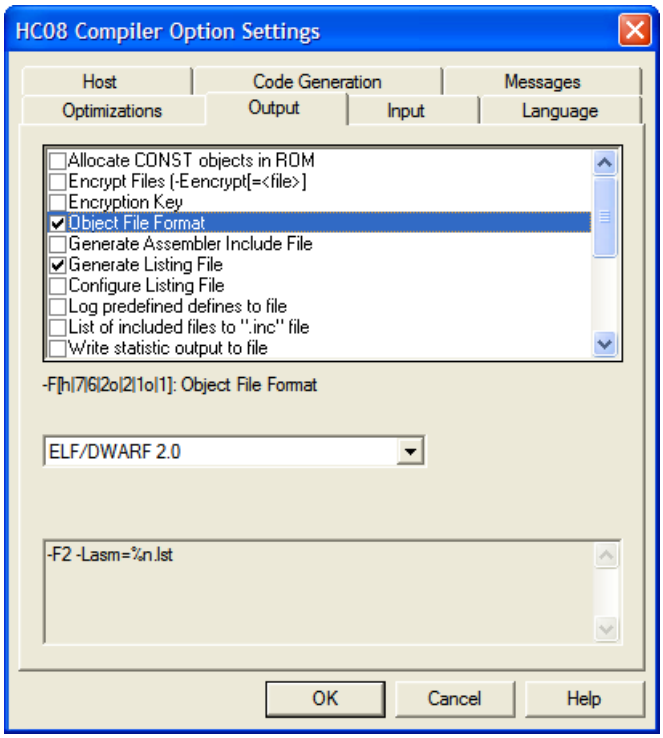

Figure 4-20. Option Settings Dialog Box

Table 4-7. Option Settings Dialog Box Tabs

Tab	Description
Optimizations	Lists optimization options
Output	Lists options related to the output file generation, such as which kind of file should be generated (list, map file, and so on)
Input	Lists options that direct how the tool handles the input file
Language	Lists options related to the programming language features used (strict ANSI-C, C++, compactC++)
Target	Lists options related to the target processor
Host	Lists options related to the host operating system
Code Generation	Lists options related to code generation (memory models, floating-point format, target processor)
Messages	Lists options that control the generation of error messages
Various	Lists options not included in the above options

Checking the application option checkbox sets the option. To obtain more detailed information about a specific option, click once on the option text to select the option and press the **F1** key or click **Help** .

NOTE

When you select options requiring additional parameters, you can open an edit box or an additional sub-window where the additional parameters are set.

4.8 Smart Control Dialog Box

Use the **Smart Control** dialog box to define the optimization level you want to apply during compilation of the input file. Five sliders are available to adjust how the source code is optimized, such as for execution speed or compact size. Refer to the table below for option descriptions.

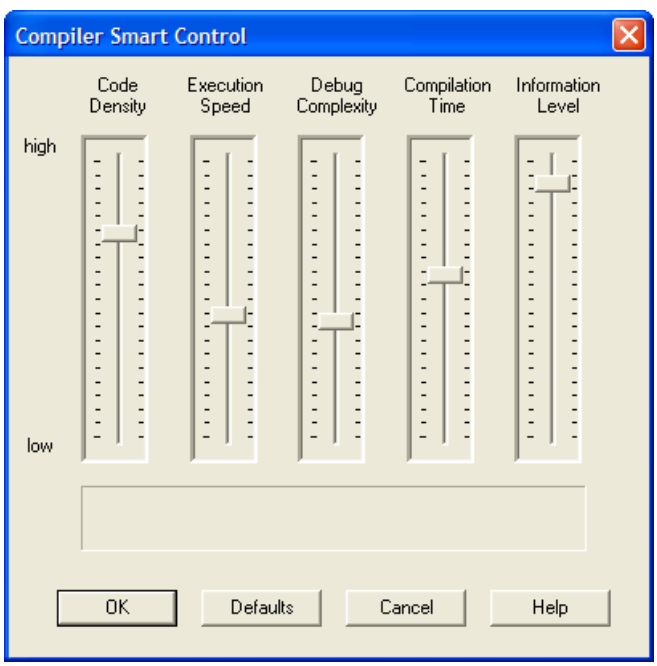

Figure 4-21. Compiler Smart Control Dialog Box

Table 4-8. Compiler Smart Control Dialog Box Controls

Slider	Description
Code Density	Displays expected code density level. A high value indicates highest code efficiency (smallest code size).
Execution Speed	Displays expected execution speed level. A high value indicates fastest execution of the code generated.
Debug Complexity	Displays expected debug complexity level. A high value indicates complex debugging. For example, assembly code corresponds directly to the high-level language code.

Table continues on the next page...

Table 4-8. Compiler Smart Control Dialog Box Controls (continued)

Slider	Description
Compilation Time	Displays expected compilation time level. A higher value indicates longer compilation time to produce the object file, often due to the manys optimizations applies to the code.
Information Level	Displays level of information messages displayed during compiler session. A high value indicates a verbose compiler behavior. For example, it informs with warnings and information messages.

The first four sliders in this panel are directly linked. That is, moving one slider updates the positions of the other three, according to the adjustment.

The command line automatically updates with the options set in accordance with the settings of the different sliders.

4.9 Message Settings Dialog Box

Use the **Message Settings** dialog box to map messages to a different message class.

Some buttons in the panel may be disabled. (For example, if an option cannot be moved to an Information message, the **Move to: Information** button is disabled.) The following table describes the buttons available in this dialog box.

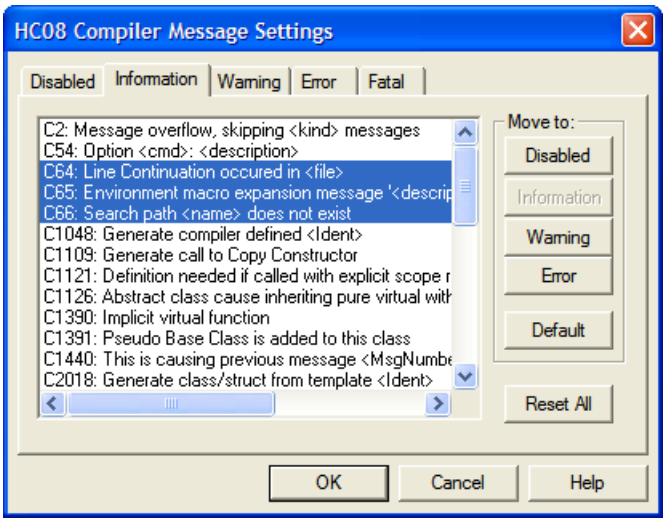

Figure 4-22. Message Settings Dialog Box

Table 4-9. Message Settings Dialog Box Buttons

Button	Description
Move to: Disabled	Disables selected messages. (Messages no longer occur.)
Move to: Information	Changes the selected messages to information messages.
Move to: Warning	Changes the selected messages to warning messages.
Move to: Error	Changes the selected messages to error messages.
Move to: Default	Changes the selected messages to their default message type.
Reset All	Resets all messages to their default message type.
OK	Exits this panel and accepts the changes made.
Cancel	Exits this panel without accepting the changes made.
Help	Displays online help about this panel.

A panel is available for each error message class. The content of the list box depends on the selected panel. The following table lists the message group definitions.

Table 4-10. Message Group Definitions

Message Group	Description
Disabled	Lists all disabled messages. Messages displayed in the list box will not be displayed by the tool.
Information	Lists all information messages. Information messages describe the actions taken by the tool.
Warning	Lists all warning messages. When a warning message is generated, input file processing continues.
Error	Lists all error messages. When an error message is generated, input file processing continues.

Table continues on the next page...

Table 4-10. Message Group Definitions (continued)

Message Group	Description
Fatal	Lists all fatal error messages. When a fatal error message is generated, input file processing stops immediately. Fatal messages cannot be changed and are only listed to provide access to context help.

Each message has its own prefix (for example, `C' for Compiler messages, `A' for Assembler messages, `L' for Linker messages, `M' for Maker messages, `LM' for Libmaker messages) followed by a 4- or 5-digit number. This number allows an easy search for the message both in the manual or on-line help.

This section covers the following topics:

- Changing Class Associated with Message
- Retrieving Information about Error Message

4.9.1 Changing Class Associated with Message

Use the buttons located on the right hand side of the panel to configure your own message mapping among the different message classes. Each button refers to a message class. To change the class associated with a message, select the message in the list box and then click the button associated with the class to which you want to move the message.

To define a warning message as an error message:

1. Click the **Warning** panel to display the list of all warning messages in the list box.
2. Select the message you want to change in the list box.
3. 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 active only when messages that can be moved are selected. When you select a message which cannot be moved to a specific group, the corresponding **Move to** button for that group is disabled (grayed).

To save your changes to the message mappings, click **OK** to close the **Message Settings** panel. Clicking **Cancel** discards your changes and closes the panel.

4.9.2 Retrieving Information about Error Message

You can access information about each message displayed in the list box. Select the message in the list box and then click **Help** or press the F1 key. An information box opens which contains a more detailed description of the error message, as well as a small example of code that may have generated the error message. If several messages are selected, help information for the first is shown. When no message is selected, pressing the F1 key or the **Help** button shows the help information for this panel.

4.10 About Dialog Box

Open the **About** dialog box by selecting **Help > About** . The **About** window contains information regarding your tool. The current directory and the versions of subparts of the application are also shown. The main version displays separately at the top of the window.

Use the **Extended Information** button to get license information about all software components in the same directory as that of the executable file.

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

NOTE

During processing, the **Extended Information** button cannot request information regarding software components. This information displays only if the tool is inactive.

4.11 Specifying Input File

There are different ways to specify the input file. During compilation, the configuration established in the different dialog boxes sets the options. Before compiling a file, make sure you have associated a working directory with your editor.

The topics covered here are as follows:

4.11.1 Methods of Compilation

There are several different methods for compiling input files. The following table lists different compilation methods.

Table 4-11. Compilation Methods

Method	Description
Use the Command Line in the Toolbar to Compile	Use the command line to compile a new file or to open a previously compiled file.
Compile a New File	Enter a new filename and additional compiler options in the command line. The specified file compiles as soon as you click **Compile** in the toolbar or press the Enter key.
Compile an Existing File	Display the previously executed command using the arrow on the right side of the command line. Select a command by clicking on it. It appears in the command line. The specified file compiles as soon as you click **Compile** in the toolbar.
Use File > Compile	When you select **File > Compile** , a standard open file dialog appears. Use this to browse to the file you want to compile. The selected file compiles as soon as you click **Open**.
Use Drag and Drop	Drag a filename from an external application (for example the File Manager/Explorer) and drop it into the compiler window. The dropped file compiles as soon as you release the mouse button. Files with the `*.ini` extension load immediately as configuration files and do not compile. Use one of the other methods to compile a C file with the `*.c` extension.

4.11.2 Message/Error Feedback

Check for errors or warnings after compilation by using one of several methods. The following listing lists the format of the error messages.

Listing: Format 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>
```

The following listing shows a typical error message.

Listing: Example of an Error Message

```
>> in "C:\DEMO\fibo.c", line 30, col 10, pos 428
   EnableInterrupts

   WHILE (TRUE) {

         (

INFORMATION C4000: Condition always TRUE
```

Refer also the -WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode and -WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode compiler options for different message formats.

4.11.3 Use Information from Compiler Window

Once a file compiles, the compiler window content area displays the list of all detected errors or warnings.

Use your editor to open the source file and correct the errors.

4.11.4 Working with User-Defined Editor

Configure the edito r you want to use for message/error feedback in the **Configuration** panel before you begin the compile process. Once a file compiles, double-click on an error message. The selected editor automatically activates and points to the line containing the error.

Chapter 5
Environment

This chapter describes all the environment variables that the compiler uses. Some environment variables are also used by other tools. Consult the tool manuals for specific variable use information.

The major sections in this chapter are:

Use environment variables to set environment parameters. Specify your environment using one of the following methods:

- (Recommended Method) The current project file with the [**Environment Configuration**] panel. Specify this file on tool startup using the `-Prod` option (-Prod: Specify Project File at Startup).
- An optional `default.env` file in the current directory. CodeWarrior IDE tools support backward compatibility for this file. Specify the filename using the `ENVIRONMENT` variable (ENVIRONMENT: Environment File Specification). Using the `default.env` file is not recommended.
- Setting environment variables on system level (DOS level). This method is not recommended.

Use the following syntax for setting an environment variable:

```
Parameter:  <KeyName>=<ParamDef>
```

NOTE

Normally *no* white space is allowed in the definition of an environment variable.

Listing: Setting the GENPATH Environment Variable

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

Define parameters using one of several methods:

- Using system environment variables supported by your operating system.

- Putting the definitions into the actual project file in the [Environment Variables] section.
- Putting the definitions in the `default.env` file in the default directory.

NOTE

The maximum length of environment variable entries in the `default.env` file is 4096 characters.

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

NOTE

Set the default directory mentioned above using the `DEFAULTDIR` system environment variable (DEFAULTDIR: Default Current Directory).

When looking for an environment variable, all programs first search the system environment, then the `default.env` file, and finally the global environment file defined by `ENVIRONMENT`. If no definition is found, the tools assume a default value.

NOTE

You can also change the environment using the `-Env` option (-Env: Set Environment Variable).

NOTE

Make sure that there are no spaces at the end of any environment variable declaration.

5.1 Current Directory

The *current directory* is the base search directory in which the tool begins a search for files (for example, for the `default.env` file). Either the operating system or the program determines the current directory of a tool.

- For UNIX operating systems, the current directory is the directory in which the binary file is located when launched.
- For Microsoft Windows-based operating systems, the current directory is defined as follows:
  - If the tool is launched using the File Manager or Explorer, the current directory is the location of the launched executable.
  - If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon.
  - If the tool is launched by another launching tool with its own current directory specification (for example, an editor), the current directory is the one specified by the launching tool (for example, current directory definition).
- For the tools, the location of the local project file defines the current directory. Changing the current project file also changes the current directory if the new project file is in a different directory.

NOTE

Browsing for a C file does not change the current directory.

Specify a default current directory using the `DEFAULTDIR` environment variable (DEFAULTDIR: Default Current Directory).

Display the current directory and other information using the `-V` compiler option (-V: Prints the Compiler Version) or the **About** dialog.

5.2 Environment Macros

You can use macros in your environment settings, as shown in the following listing.

Listing: Using Macros for Setting Environment Variables

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

In the example above, `TEXTPATH` expands to `C:\test\txt` and `OBJPATH` expands to `C:\test\obj`. You can use `$()` or `${}`. However, you must define the referenced variable.

You may also use special variables when using macros. Always surround special variables with brackets `{}`. Special variables are case-sensitive. In addition, the variable content contains the directory separator ` \'. The special variables are:

- { `Compiler`}

Path of the executable one directory level up, if the executable file is `C:\Freescale\prog\linker.exe`, and the variable is `C:\Freescale\`.

- { `Project`}

  Path of the current project file. Use this special variable if the current project file is `C:\demo\project.ini`, and the variable contains `C:\demo\`.

- { `System`}

  Path of your Windows system installation, for example, `C:\WINNT\`.

5.3 Global Initialization File (mcutools.ini)

All tools store some global data in the `mcutools.ini` file. The tool first searches for `mcutools.ini` in the tool directory (path of the executable). If `mcutools.ini` is not found in the tool directory, the tool looks for `mcutools.ini` in the Microsoft Windows installation directory (for example, `C:\WINDOWS`).

Listing: Typical Global Initialization File Locations

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

If you start the tool in the `D:\INSTALL\prog` directory, the tool uses the project file located in the same directory as the tool (`D:\INSTALL\prog\mcutools.ini`).

If you start the tool some other way, the tool uses the project file in the Windows directory (`C:\WINDOWS\mcutools.ini`).

Global Configuration File Entries documents the sections and entries you can include in `mcutools.ini`.

5.4 Local Configuration File

The configuration file stores all configuration properties. Different applications use the same configuration file.

The shell uses the `project.ini` configuration file in the current directory. When the shell uses the same file as the compiler, the shell writes and maintains the Editor Configuration used by the compiler. Apart from this, the compiler can use any filename for the project

file. The configuration file has the same format as the windows *.ini files. The compiler stores its own entries with the same section name as those in the global mcutools.ini file. The compiler backend is encoded into the section name, so that a different compiler backend can use the same file without any overlapping. Different versions of the same compiler use the same entries. When you must store options available in only one compiler version in the configuration file, you must maintain a separate configuration file for each different compiler version. If no incompatible options are enabled, you may use the same file for both compiler versions.

Be aware of these rules when changing the directory or making changes to the configuration file:

- The current directory is always the directory in which the configuration file is located.
- Loading a configuration file in a different directory changes the current directory.
- Changing the current directory reloads the entire default.env file.
- Loading or storing a configuration file reloads the options in the environment variable COMPOPTIONS and adds the options to the project options. This becomes an issue when different default.env files, containing incompatible options in the COMPOPTIONS variable, exist in different directories.
- Loading a project using the default.env adds the options in COMPOPTIONS to the configuration file.
  - Storing this configuration in a different directory which contains a default.env with incompatible options adds the options to default.env. The compiler remarks the inconsistency.
  - Use the option settings dialog box to remove the option from the configuration file, or remove the option from the default.env with the shell or a text editor.

At startup, use one of two methods to load a configuration:

- Use the -Prod: Specify Project File at Startup command line option
- Use the project.ini file in the current directory.

Using the -Prod option uses the directory which contains the project file as the current directory. Specifying the directory when using the -Prod option loads the project.ini file in this directory.

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

5.5 Paths

A path list is a list of directory names separated by semicolons. Declare path names using the following Extended Backus-Naur Form (EBNF) syntax . Most environment variables contain path lists to direct the compiler to the file location as shown in the *Listing: Environment Variable Path List with Four Possible Path*s.

Listing: EBNF Path Syntax

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

Listing: Environment Variable Path List with Four Possible Paths

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

An asterisk (`*`) preceding a directory name instructs the program to recursively search the entire directory tree for a file, not just the given directory itself. The program searches the directories in the order of appearance in the path list.

Listing: Setting an Environment Variable using Recursive Searching

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

NOTE

This procedure uses some DOS environment variables (like `GENPATH` or `LIBPATH`).

If you work with the CodeWright editor, you can set the environment using a `<project>.pjt` file in your project directory. This enables you to have different projects in different directories, each with its own environment.

NOTE

When using WinEdit, do *not* set the system environment variable DEFAULTDIR: Default Current Directory. If you do so, and this variable does not contain the project directory given in WinEdit's project configuration, files might not be placed where you expect them to be.

5.6 Line Continuation

Specify an environment variable in an environment file (`default.env`) over several lines using the line continuation character `'\'` (Refer the following listing).

Listing: Specifying an Environment Variable using Line Continuation Characters

```
OPTIONS=\ -W2 \ -Wpd
```

This is the same as:

```
OPTIONS=-W2 -Wpd
```

However, this feature may not work well used together with paths. For example, the following code is a valid use of the continuation character:

```
GENPATH=.\

TEXTFILE=.\txt
```

However the compiler understands these instructions as:

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

To avoid such problems, use a semicolon (;) at the end of a path if using a continuation character (\) at the end of the line, as shown in the following listing.

Listing: Using a Semicolon to Allow a Multiple Line Environment Variable

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

5.7 Environment Variable Details

The remainder of this chapter describes each of the possible environment variables. The following table lists these description topics in their order of appearance for each environment variable.

Table 5-1. Environment Variables-Documentation Topics

Topic	Description
Tools	Lists tools that use this variable.
Synonym	Lists any synonyms that exist for the environment variable. Synonyms may be used for older releases of the Compiler and will be removed in the future. A synonym has lower precedence than the environment variable.
Syntax	Specifies the syntax of the option in EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default	Shows the default setting for the variable or none.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and the effects of the variable when possible. The example shows an entry in the `default.env` for a PC.
See also	Names related sections.

The environment variables included here are as follows:

5.7.1 COMPOPTIONS: Default Compiler Options

Tools

Compiler

Synonym

HICOMPOPTIONS

Syntax

```
COMPOPTIONS={<option>}
```

Arguments

`<option>`: Compiler command-line option

Default

None

Description

Set this environment variable to instruct the Compiler to append its contents to the command line each time a file is compiled. Use this environment variable to specify global options that should always be set. This frees you from having to specify them for every compilation.

NOTE

We do not recommend using this environment variable if the Compiler used is version 5.x, because it adds the specified options to the options stored in the `project.ini` file.

Listing: Setting Default Values for Environment Variables (Not Recommended)

```
COMPOPTIONS=-W2 -Wpd
```

See also

Compiler Options

5.7.2 COPYRIGHT: Copyright Entry in Object File

Tools

Compiler, Assembler, Linker, or Librarian

Synonym

None

Syntax

```
COPYRIGHT=<copyright>
```

Arguments

< `copyright`>: copyright entry

Default

None

Description

Each object file contains an entry for a copyright string. The decode retrieves this information from the object files.

Example

```
COPYRIGHT=Copyright by Freescale
```

See also

environmental variables :

- USERNAME: User Name in Object File
- INCLUDETIME: Creation Time in Object File

5.7.3 DEFAULTDIR: Default Current Directory

Tools

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

Synonym

None

Syntax

```
DEFAULTDIR=<directory>
```

Arguments

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

Default

None

Description

Specifies the default directory for all tools. All the tools indicated above take the specified directory as their current directory instead of the directory defined by the operating system or launching tool.

NOTE

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

Specify the default directory for all tools in the CodeWarrior suite:

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

See also

Current Directory

Global Initialization File (mcutools.ini)

5.7.4 ENVIRONMENT: Environment File Specification

Tools

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

Synonym

`HIENVIRONMENT`

Syntax

```
ENVIRONMENT=<file>
```

Arguments

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

Default

None

Description

This variable is specified on a system level. The application looks in the current directory for an environment file named `default.env`. Use `ENVIRONMENT` (for example, set in the `autoexec.bat` (DOS) or `*.cshrc` (UNIX)), to specify a different filename.

NOTE

This system level (global) environment variable cannot be specified in a default environment file (`default.env`).

Example

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

5.7.5 ERRORFILE: Error Filename Specification

Tools

Compiler, Assembler, Linker, or Burner

Synonym

None

Syntax

```
ERRORFILE=<filename>
```

Arguments

`<filename>`: filename with possible format specifiers

Description

The `ERRORFILE` environment variable specifies the name for the error file.

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`).

A notification box displays in the event of an improper error filename.

Examples

```
ERRORFILE=MyErrors.err
```

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

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

Writes all errors into the errors file in the `\tmp` directory.

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

Writes all errors into a file with the same name as the source file, but with the `*.err` extension, into the same directory as the source file. Using this format, compiling a file named `test.c` in the `sources` directory (`\sources\test.c`) generates an error list file named `test.err` in the `sources` directory (`\sources\test.err`).

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

Writes all errors into a file in the `dir1` directory. Compiling a source file named `test.c` generates an error list file named `\dir1\test.err`.

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

Writes all errors into a text file in the same directory as the compiled file. Compiling a source file named `\dir1\dir2\test.c` generates an error list file with the name `\dir1\dir2\errors.txt`.

Unless you set the `ERRORFILE` environment variable, the compiler writes errors to the `EDOUT` file in the current directory.

5.7.6 GENPATH: #include "File" Path

Tools

Compiler, Linker, Decoder, Debugger, or Burner

Synonym

HIPATH

Syntax

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

Arguments

<path>: Paths separated by semicolons, without spaces

Default

Current directory

Description

Including a header file with double quotes makes the Compiler search first in the current directory, then in the directories listed by GENPATH, and finally in the directories listed by LIBRARYPATH.

NOTE

If a directory specification in this environment variable starts with an asterisk (*), the compiler searches the whole directory tree recursively depth first, i.e., all subdirectories and *their* subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.

Example

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

See also

LIBRARYPATH: `include <File>' Path environment variable

5.7.7 INCLUDETIME: Creation Time in Object File

Tools

Compiler, Assembler, Linker, or Librarian

Synonym

None

Syntax

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

Arguments

ON: Include time information into object file

OFF: Do not include time information into object file

Default

ON

Description

Each object file contains a time stamp indicating the creation time and date as strings. When a tool creates a new file, the file gets a new time stamp entry.

This behavior may be undesired if you must perform a binary file compare for Software Quality Assurance reasons. Even if the information in two object files is the same, the files do not match exactly as the time stamps are not identical. To avoid such problems, set this variable to OFF. When set to OFF, the time-stamp strings in the object file for date and time are "none" in the object file.

Retrieve the time stamp from the object files using the decoder.

Example

```
INCLUDETIME=OFF
```

See also

- COPYRIGHT: Copyright Entry in Object File
- USERNAME: User Name in Object File

5.7.8 LIBRARYPATH: `include <File>' Path

Tools

Compiler, Burner, Linker, Decoder

Synonym

LIBPATH

Syntax

```
LIBRARYPATH={<path>}
```

Arguments

< path>: Paths separated by semicolons, without spaces

Default

Current directory

Description

Including a header file with double quotes makes the Compiler search first in the current directory, then in the directories given by GENPATH, and finally in the directories given by LIBRARYPATH (see GENPATH: #include "File" Path LIBRARYPATH: `include <File>' Path).

NOTE

If a directory specification in this environment variable starts with an asterisk (*), the compiler searches the whole directory tree recursively depth first, i.e., all subdirectories and *their* subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.

Example

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

See also

GENPATH: #include "File" Path environment variable

USELIBPATH: Using LIBPATH Environment Variable

Input Files

5.7.9 OBJPATH: Object File Path

Tools

Compiler, Linker, Decoder, Debugger, Burner

Synonym

None

Syntax

```
OBJPATH=<path>
```

Default

Current directory

Arguments

<path>: Path without spaces

Description

The Compiler places generated object files into the directory specified by OBJPATH. If this environment variable is empty or does not exist, the compiler places the object file in the same directory as the source code.

If the Compiler tries to generate an object file into the directory specified by this environment variable but fails (for example, because the file is locked), the Compiler issues an error message.

If a tool looks for an object file, it first checks for an object file specified by this environment variable, then in GENPATH, and finally in HIPATH (refer GENPATH: #include "File" Path, HIPATH).

Example

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

See also

Output Files

5.7.10 TEXTPATH: Text File Path

Tools

Compiler, Linker, Decoder

Synonym

None

Syntax

```
TEXTPATH=<path>
```

Arguments

<path>: Path without spaces

Default

Current directory

Description

The Compiler places generated text files into the directory specified by TEXTPATH. If this environment variable is empty or does not exist, the compiler stores the text file into the current directory.

Example

```
TEXTPATH=\sources\txt
```

See also

Output Files

Compiler options :

- -Li: List of Included Files to ".inc" File
- -Lm: List of Included Files in Make Format
- -Lo: Append Object File Name to List (enter [<files>])

5.7.11 TMP: Temporary Directory

Tools

Compiler, Assembler, Linker, Debugger, Librarian

Synonym

None

Syntax

```
TMP=<directory>
```

Arguments

<directory>: Directory to be used for temporary files

Default

None

Description

The compiler uses the ANSI function, tmpnam(), to create temporary files. This library function stores the temporary files in the directory specified by the TMP environment variable. If the variable is empty or does not exist, the compiler uses the current directory. Check this variable if you get the error message "Cannot create temporary file".

NOTE

This is a system level (global) environment variable. It cannot be specified in a default environment file (default.env).

Example

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

See also

Current Directory

5.7.12 USELIBPATH: Using LIBPATH Environment Variable

Tools

Compiler, Linker, or Debugger

Synonym

None

Syntax

```
USELIBPATH=(OFF|ON|NO|YES)
```

Arguments

ON, YES: The Compiler uses the environment variable LIBRARYPATH to look for system header files <*.h>.

NO, OFF: The Compiler does not use the environment variable LIBRARYPATH.

Default

ON

Description

This environment variable allows a flexible usage of the LIBRARYPATH environment variable as the LIBRARYPATH variable might be used by other software (for example, version management PVCS).

Example

```
USELIBPATH=ON
```

See also

LIBRARYPATH: `include <File>' Path environment variable

5.7.13 USERNAME: User Name in Object File

Tools

Compiler, Assembler, Linker, Librarian

Synonym

None

Syntax

```
USERNAME=<user>
```

Arguments

<user>: Name of user

Default

None

Description

Each object file contains an entry identifying the creator the object file. Retrieve this information from the object files by using the decoder.

Example

```
USERNAME=The Master
```

See also

environment variables :

- COPYRIGHT: Copyright Entry in Object File
- INCLUDETIME: Creation Time in Object File

Chapter 6
Files

This chapter describes the input and output files that the Compiler uses, and file processing. It has the following sections:

- Input Files
- Output Files
- File Processing

6.1 Input Files

This section describes the following input files:

- Source Files
- Include Files

6.1.1 Source Files

The compiler's frontend takes any file as input. It does not require the filename to have a special extension. However, it is suggested that all your source filenames use the `*.c` extension and that all header files use the `*.h` extension. The Compiler searches for so *urce files first in the*Current Directory *and then in the* `GENPATH` *directory (refer* GENPATH: #include "File" Path).

6.1.2 Include Files

The search for include files is governed by two environment variables: GENPATH: #include "File" Path and LIBRARYPATH: `include <File>' Path. Include files are included using double quotes or angle brackets.

When you use double quotes to include files (`#include "test.h"`), the compiler searches first in the current directory, then in the directory specified by the `-I` option (refer -I: Include File Path), then in the directories given in the GENPATH environment variable (refer *GENPATH: #include "File" Path*), and finally in those listed in the LIBPATH or LIBRARYPATH: ` include <File>' Path environment variable. Set the current directory using the IDE, the Program Manager, or the DEFAULTDIR environment variable (refer DEFAULTDIR: Default Current Directory).

When you use angle brackets to include files (`#include <stdio.h>`), the compiler searches first in the current directory, then in the directory specified by the `-I` option, and then in the directories given in LIBPATH or LIBRARYPATH. Set the current directory using the IDE, the Program Manager, or the DEFAULTDIR environment variable.

6.2 Output Files

This section describes the following output files:

- Object Files
- Error Listing

6.2.1 Object Files

After successful compilation, the Compiler generates an object file containing the target code as well as some debugging information. The compiler writes this file to the directory listed in the OBJPATH environment variable (refer OBJPATH: Object File Path). If that variable contains more than one path, the compiler writes the object file in the first listed directory. If this variable is not set, the compiler writes the object file in the same directory as the source code file. Object files always get the extension `*.o`.

6.2.2 Error Listing

If the Compiler detects any errors, it creates an error listing file named `err.txt` instead of an object file. The compiler generates this file in the same directory as the source code file (refer ERRORFILE: Error Filename Specification).

The open Compiler window displays the full path of all header files read. After successful compilation the window also displays the number of code bytes generated and the number of global objects written to the object file.

If you start the Compiler from an IDE (with `'%f'` given on the command line) or CodeWright (with `'%b%e'` given on the command line), the Compiler does not produce the `err.txt` error file. Instead, the Compiler writes the error messages in a special format in a file called `EDOUT` using the default Microsoft format. You may use the CodeWrights' *Find Next Error* command to display both the error positions and the error messages.

6.2.2.1 Interactive Mode (Compiler Window Open)

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

If you do not set `ERRORFILE`, the Compiler generates a default file named `err.txt` in the current directory.

6.2.2.2 Batch Mode (Compiler Window Not Open)

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

If you do not set `ERRORFILE`, the Compiler generates a default file named `EDOUT` in the current directory.

6.3 File Processing

The following figure shows how file processing occurs with the Compiler:

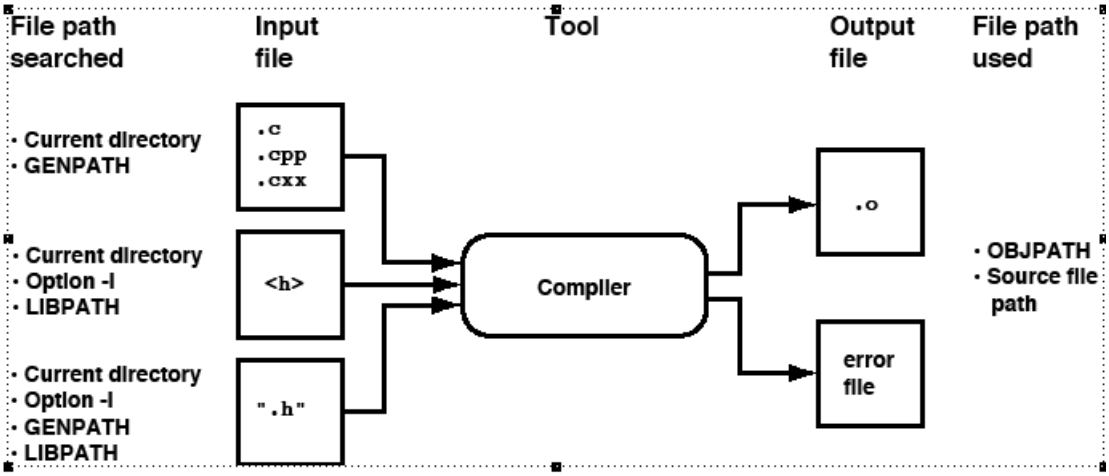

Figure 6-1. Files Used with the Compiler

Chapter 7
Compiler Options

This chapter describes the arguments that modify how the compiler reads source files, processes them, and outputs object code. It consists of the following major sections:

- Option Recommendations : Suggests some compiler settings that produce efficient code or make it easier to debug.
- Compiler Option Details : Describes the layout and format of the compiler command-line options covered in the remainder of the chapter.

The Compiler provides a number of options that control its operation. Options consist of a minus sign or dash (-) prefix, followed by one or more letters or digits. The Compiler treats anything lacking the prefix as a source file to compile. You can specify Compiler options on the command line or in the COMPOPTIONS variable. Specify each Compiler option only once per compilation.

Command line options are not case-sensitive, for example, "-Li" is the same as "-li".

NOTE

It is not possible to combine options from different groups. For example, "-Cc -Li" *cannot* be abbreviated by the terms "-Cci" or "-CcLi".

You can also use the GUI to set the Compiler options, as shown in the following figure.

NOTE

If you use the GUI to set the Compiler options, do not use the COMPOPTIONS environment variable. The Compiler stores the options in the project.ini file, not in the default.env file.

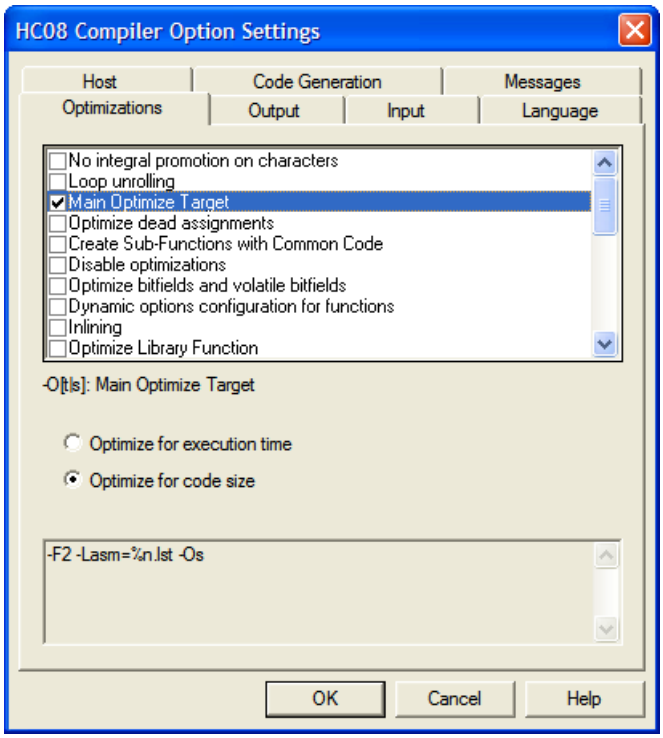

Figure 7-1. HC08 Compiler Option Settings Dialog Box

Use the **Message Settings** dialog box, shown in the following figure, to move messages (`-Wmsg` options).

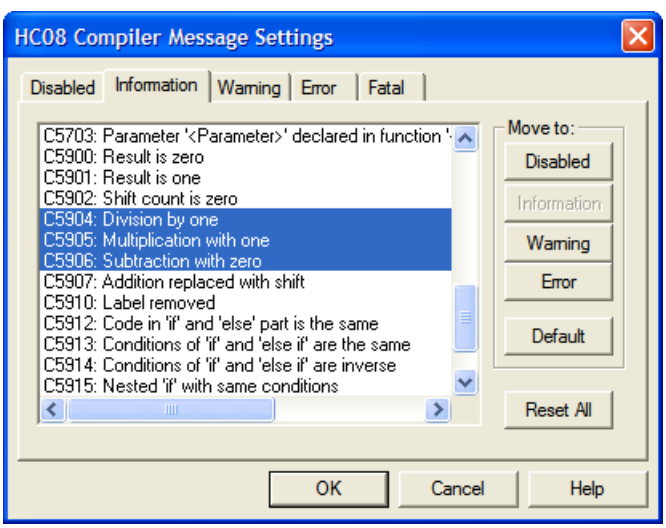

Figure 7-2. Message Settings Dialog Box

7.1 Option Recommendations

Depending on the compiled source code files, each Compiler option has advantages or disadvantages. We recommend the following:

- When using the HIWARE Object-file Format and the -Cc: Allocate Const Objects into ROM compiler option, specify `ROM_VAR` in the Linker parameter file.
- -Wpd: Error for Implicit Parameter Declaration
- -Or: Register Optimization whenever available or possible

The default configuration enables most options in the Compiler. If the default options cause problems in your code (for example, they make the code hard to debug), disable the options (these options usually have the `-On` prefix). Candidates for such options are peephole options.

Some options produce more code for some functions than for others (for example, -Oi: Inlining or -Cu: Loop Unrolling). On the other hand, inlining functions, while producing more code, also make it execute faster because there's not overhead in performing a function call. Vary your option choices to find the best result for each function.

To get the best results for each function, compile each module with the `-OdocF` option (refer -OdocF: Dynamic Option Configuration for Functions). An example of this option is `-OdocF="-Or"`.

For compilers with the ICG optimization engine, the following option combination provides the best results:

```
-Ona -OdocF="-Onca|-One|-Or"
```

7.2 Compiler Option Details

This section describes the option groups in detail, plus the layout and format of each option. This section covers the following topics:

- Option Groups
- Option Scopes
- Option Detail Description

7.2.1 Option Groups

Compiler options are grouped by:

- HOST

- LANGUAGE
- OPTIMIZATIONS
- CODE GENERATION
- OUTPUT
- INPUT
- MESSAGES
- VARIOUS
- STARTUP

NOTE

The option in the *Various* group do not appear in the **HC08 Compiler Option Settings** dialog box, however the options are displayed as a part of the compiler option list when using the `-H` option.

Refer to the following table for description of each of the groups.

Specify options in the STARTUP group on the command line to start the tool. Options in the STARTUP group cannot be specified interactively.

Table 7-1. Compiler Option Groups

Group	Description
HOST	Lists options related to the host
LANGUAGE	Lists options related to the programming language (for example, ANSI-C)
OPTIMIZATIONS	Lists optimization options
OUTPUT	Lists options related to the output files generation (i.e., which kind of file to generate)
INPUT	Lists options related to the input file
CODE GENERATION	Lists options related to code generation (memory models, float format, etc.)
MESSAGES	Lists options controlling error message generation
VARIOUS	Lists various options
STARTUP	Lists options specified only on tool startup

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

NOTE

Not all command line options are accessible through the property sheets as they have a special graphical setting (for example, the option to set the type sizes).

7.2.2 Option Scopes

Each option has a scope. Refer the following table.

Table 7-2. Option Scopes

Scope	Description
Application	Set the option for all files (Compilation Units) of an application. A typical example is an option to set the memory model. Mixing object files has unpredictable results.
Compilation Unit	Set this option differently for each compilation unit for an application. Mixing objects in an application is possible.
Function	Set the option differently for each function. Use a function option with the option: `-OdocF= "<option>".`
None	The option scope is not related to a specific code part. Typical examples are the message management options.

The available options are arranged into different groups. Each group has its own sheet. The content of the list box depends on the selected sheets.

7.2.3 Option Detail Description

The remainder of this section describes the Compiler options in alphabetical order. The following table describes the information available for each option.

NOTE

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

Table 7-3. Compiler Option-Documentation Topics

Topic	Description
Group	Specifies HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGES or VARIOUS.
Scope	Specifies Application, Compilation 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
Defines	Lists defines related to the compiler option
Pragma	Lists pragmas related to the compiler option
Description	Provides a detailed description of the option and how to use it

Table continues on the next page...

Table 7-3. Compiler Option-Documentation Topics (continued)

Topic	Description
Example	Gives an example of usage and effects of the option where possible. Displays compiler settings, source code and Linker PRM files when applicable. The example shows an entry in the `default.env` for a PC.
See also	Names related options

7.2.3.1 Using Special Modifiers

You can use special modifiers with some options. However, some modifiers may not make sense for all options. This section describes those modifiers.

The following table lists the supported modifiers:

Table 7-4. Compiler Option Modifiers

Modifier	Description
`%p`	Path including file separator
`%N`	Filename in strict 8.3 format
`%n`	Filename without extension
`%E`	Extension in strict 8.3 format
`%e`	Extension
`%f`	Path + filename without 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)`	Use the contents of an environment variable
`%%`	Generates a single `%'

7.2.3.1.1 Example

For the following examples, the actual base filename for the modifiers is: `C:\Freescale\my demo\TheWholeThing.myExt`.

`The %p` modifier gives the path only with a file separator:

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

`Using the %N` modifier results in a filename in 8.3 format (that is, the name with only eight characters):

```
TheWhole
```

`The %n` modifier returns the whole the filename without extension:

```
TheWholeThing
```

`The %E` modifier gives the extension in 8.3 format (that is, the extension with only three characters):

```
myE
```

`Use the %e` modifier to get the whole extension:

```
myExt
```

`The %f` modifier gives the path plus the filename:

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

When the path contains a space, we recommend using `%"` or `%'`. Thus, `%"%f%"` gives (using double quotes):

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

Using `%'%f%'` gives (using single quotes):

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

The modifier `%(envVariable)` uses an environment variable. The Compiler ignores a file separator following `%(envVariable)` if the environment variable is empty or non-existent. In other words, if you set `TEXTPATH` to:

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

the Compiler replaces:

```
%(TEXTPATH)\myfile.txt
```

with:

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

But if `TEXTPATH` is empty or non-existent, the Compiler replaces:

```
%(TEXTPATH)\myfile.txt
```

with:

```
myfile.txt
```

Use the `%%` modifier to print a percent sign. Using `%e%%` gives:

```
myExt%
```

The following table lists the command-line options for the HCS08 compiler:

Table 7-5. Compiler Command-line Options

-!: Filenames are clipped to DOS Length
-AddIncl: Additional Include File
-Ansi: Strict ANSI
-ArgFile: Specify a file from which additional command line options will be read
-Asr: It is Assumed that HLI Code Saves Written Registers
-BfaB: Bitfield Byte Allocation
-BfaGapLimitBits: Bitfield Gap Limit
-BfaTSR: Bitfield Type Size Reduction
-C++ (-C++f, -C++e, -C++c): C++ Support
-Cc: Allocate Const Objects into ROM
-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers
-Ci: Bigraph and Trigraph Support
-Cn[={Vfl|Tpl|Ptm...}]: Disable compactC++ features
-Cni: No Integral Promotion on Characters
-Cppc: C++ Comments in ANSI-C
-Cq: Propagate const and volatile Qualifiers for Structs
-C[s08|08]: Generate Code for Specific HC08 Families
-CswMaxLF: Maximum Load Factor for Switch Tables
-CswMinLB: Minimum Number of Labels for Switch Tables
-CswMinLF: Minimum Load Factor for Switch Tables
-CswMinSLB: Minimum Number of Labels for Switch Search Tables
-Cu: Loop Unrolling
-Cx: Switch Off Code Generation
-D: Macro Definition
-Ec: Conversion from 'const T*' to 'T*'
-Eencrypt: Encrypt Files
-Ekey: Encryption Key
-Env: Set Environment Variable
-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format
-Fd: Double is IEEE32
-H: Short Help
-I: Include File Path
-La: Generate Assembler Include File
-Lasm: Generate Listing File
-Lasmc: Configure Listing File
-Ldf: Log Predefined Defines to File
-Li: List of Included Files to ".inc" File
-Lic: License Information
-LicA: License Information about Every Feature in Directory

Table continues on the next page...

Table 7-5. Compiler Command-line Options (continued)

-LicBorrow: Borrow License Feature
-LicWait: Wait until Floating License is Available from Floating Li- cense Server
-LI: Write Statistics Output to File
-Lm: List of Included Files in Make Format
-LmCfg: Configuration for List of Included Files in Make Format (option -Lm)
-Lo: Append Object File Name to List (enter [<files>])
-Lp: Preprocessor Output
-LpCfg: Preprocessor Output Configuration
-LpX: Stop after Preprocessor
-M (-Mb, -Ms, -Mt): Memory Model
-MMU: Enable Memory Management Unit (MMU) Support
-N: Show Notification Box in Case of Errors
-NoBeep: No Beep in Case of an Error
-NoClrVol: Do not use CLR for volatile variables in the direct page
-NoDebugInfo: Do not Generate Debug Information
-NoEnv: Do Not Use Environment
-NoPath: Strip Path Info
-O(-Os, -Ot): Main Optimization Target
-O0 : Disable Optimizations
-Obfv: Optimize Bitfields and Volatile Bitfields
-ObjN: Object File Name Specification
-Oc: Common Subexpression Elimination (CSE)
-OdocF: Dynamic Option Configuration for Functions
-Of and-Onf: Create Sub-Functions with Common Code
-Oi: Inlining
-Oilib: Optimize Library Functions
-Ol: Try to Keep Loop Induction Variables in Registers
-Ona: Disable Alias Checking
-OnB: Disable Branch Optimizer
-Onbf: Disable Optimize Bitfields
-Onbt: Disable ICG Level Branch Tail Merging
-Onca: Disable any Constant Folding
-Oncn: Disable Constant Folding in Case of a New Constant
-OnCopyDown: Do Generate Copy Down Information for Zero Values
-OnCstVar: Disable CONST Variable by Constant Replacement
-One: Disable any Low Level Common Subexpression Elimination
-OnP: Disable Peephole Optimization
-OnPMNC: Disable Code Generation for NULL Pointer to Member Check
-Ont: Disable Tree Optimizer
-OnX: Disable Frame Pointer Optimization

Table continues on the next page...

Table 7-5. Compiler Command-line Options (continued)

-Or: Allocate Local Variables into Registers
-Ous, -Ou, and -Onu: Optimize Dead Assignments
-Pe: Do Not Preprocess Escape Sequences in Strings with Absolute DOS Paths
-Pio: Include Files Only Once
-Prod: Specify Project File at Startup
-Qvtp: Qualifier for Virtual Table Pointers
-Rp (-Rpe, -Rpt): Large Return Value Type
-T: Flexible Type Management
-V: Prints the Compiler Version
-View: Application Standard Occurrence
-WErrFile: Create "err.log" Error File
-Wmsg8x3: Cut Filenames in Microsoft Format to 8.3
-WmsgCE: RGB Color for Error Messages
-WmsgCF: RGB Color for Fatal Messages
-WmsgCI: RGB Color for Information Messages
-WmsgCU: RGB Color for User Messages
-WmsgCW: RGB Color for Warning Messages
-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message 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: Maximum Number of Error Messages (enter <number>)
-WmsgNi: Maximum Number of Information Messages (enter <number>)
-WmsgNu: Disable User Messages
-WmsgNw: Maximum Number of Warning Messages (enter <number>)
-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
-Wpd: Error for Implicit Parameter Declaration
-WStdout: Write to Standard Output
-W1: Don't Print Information Messages
-W2: Do not Print INFORMATION or WARNING Messages

7.2.3.1.2 -!: Filenames are clipped to DOS Length

Group

INPUT

Scope

Compilation Unit

Syntax

```
-!
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Use this option, called cut, when compiling files copied from an MS-DOS file system. Filenames are clipped to eight characters (DOS length). The cut option truncates the filename in the following include directive:

Listing: Example of the Cut Option (-!)

```
#include "mylongfilename.h"
to:

#include "mylongfi.h"
```

7.2.3.1.3 -AddIncl: Additional Include File

Group

INPUT

Scope

Compilation Unit

Syntax

```
-AddIncl"<fileName>"
```

Arguments

`<fileName>`: name of file to include

Default

None

Defines

None

Pragmas

None

Description

This option includes the specified file at the beginning of the compilation unit. It has the same effect as writing the file at the beginning of the compilation unit using double quotes (`"..."`):

```
#include "my headerfile.h"
```

Refer the following listing for the `-AddIncl` compiler option syntax to include the above header file.

Listing: Syntax Example for Including a Header File

```
-AddIncl"my headerfile.h"
```

See also

-I: Include File Path compiler option

7.2.3.1.4 -Ansi: Strict ANSI

Group

LANGUAGE

Scope

Function

Syntax

```
-Ansi
```

Arguments

None

Default

None

Defines

__STDC__

Pragmas

None

Description

The -Ansi option forces the Compiler to follow strict ANSI C language conversions. When you specify -Ansi, the Compiler refuses all non-ANSI-compliant keywords (for example, __asm, __far and __near), and generates an error.

The ANSI-C compiler also does not allow C++ style comments (those started with //). To allow C++ comments, even with -Ansi set, set the -Cppc: C++ Comments in ANSI-C compiler option.

The asm keyword is also not allowed when -Ansi is set. To use inline assembly, even with -Ansi set, use __asm instead of asm.

The Compiler defines __STDC__ as 1 if this option is set, or as 0 if this option is not set.

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

Group

HOST

Scope

Function

Syntax

-ArgFile<filename>

Arguments

<filename>: Specify the file containing the options to be passed in the command line.

Description

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

Example

Considering that a file named option.txt is used and that it contains the "-Mt" option then chc08.exe -ArgFileoption.txt command line will be equivalent to chc08.exe -Mt.

7.2.3.1.6 -Asr: It is Assumed that HLI Code Saves Written Registers

Group

CODE GENERATION

Scope

Function

Syntax

```
-Asr
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option set, the compiler assumes that registers touched in HLI are saved or restored in the HLI code as well. If this option is not set, the compiler saves and restores the registers as needed.

Listing: Sample Source Code for the Two Following Examples

```
void bar(char);
char PORT;

void myfun(void) {

  PORT = 4;

  asm {

    lda   #4

    sta   PORT

  }

  bar(4);

}
```

Listing: Code Sample Without the -Asr Option

```
  4:    PORT = 4;
0000 a604             LDA   #4

0002 c70000           STA   PORT

  5:    __asm {

  6:    lda   #4

0005 a604             LDA   #4

  7:    sta   PORT

0007 c70000           STA   PORT

  8:  }

  9:    bar(4);

000a a604             LDA   #4

000c cc0000           JMP   bar
```

With the `-Asr` option set (as shown in the following listing), the compiler assumes that the A register is the same as before the `__asm` block. However, in our example we do NOT save or restore the A register, so the compiler generates incorrect code.

Listing: Code Sample With the -Asr Option

```
  4:    PORT = 4;
0000 a604             LDA   #4
```

```
0002 c70000           STA   PORT
  5:    __asm {
  6:      lda #4
0005 a604             LDA   #4
  7:      sta PORT
0007 c70000           STA   PORT
  8:    }
  9:    bar(4);
000a cc0000           JMP   bar
```

7.2.3.1.7 -BfaB: Bitfield Byte Allocation

Group

CODE GENERATION

Scope

Function

Syntax

```
-BfaB(MS|LS)
```

Arguments

MS: Most significant bit in byte first (left to right)

LS: Least significant bit in byte first (right to left)

Default

```
HC08: -BfaBLS
```

Defines

```
__BITFIELD_MSWORD_FIRST__

__BITFIELD_LSWORD_FIRST__

__BITFIELD_MSBYTE_FIRST__

__BITFIELD_LSBYTE_FIRST__

__BITFIELD_MSBIT_FIRST__
```

__BITFIELD_LSBIT_FIRST__

Pragmas

None

Description

Normally, bit allocation in byte bitfields proceeds from the least significant bit to the most significant bit. This produces less code overhead in the case of partially-allocated byte bitfields.

Example

The following listing uses the default condition and uses the three least significant bits.

Listing: Changing the Allocation Order Example

```
struct {unsigned char b: 3; } B;   // the default is using the 3 least significant bits
```

This allows a mask operation without any shift to access the bitfield.

To change this allocation order, use the -BfaBMS or -BfaBLS options shown in the following listing.

Listing: Changing the Allocation Order Example

```
struct {
  char b1:1;

  char b2:1;

  char b3:1;

  char b4:1;

  char b5:1;

} myBitfield;

  7                   0

  --------------------

  |b1|b2|b3|b4|b5|####|   (-BfaBMS)

  --------------------

  7                   0

  --------------------

  |####|b5|b4|b3|b2|b1|   (-BfaBLS)

  --------------------
```

See also

Bitfield Allocation

7.2.3.1.8 -BfaGapLimitBits: Bitfield Gap Limit

Group

CODE GENERATION

Scope

Function

Syntax

```
-BfaGapLimitBits<number>
```

Arguments

<number>: positive number, there should be less than <number> bits in the gap (that is, at most <number>-1 bits)

Default

1

Defines

None

Pragmas

None

Description

The bitfield allocation tries to avoid crossing a byte boundary whenever possible. To achieve optimized accesses, the compiler may insert some padding or gap bits to reach this. This option enables you to affect the maximum number of gap bits allowed.

Example

In the example shown in the listing below, assume that you have specified a 3-bit maximum gap, that is, -BfaGapLimitBits3.

Listing: Bitfield Allocation

```
struct {
  unsigned char a: 7;
```

```
  unsigned char b: 5;

unsigned char c: 4;
} B;
```

The compiler allocates `struct B` with 3 bytes. First, the compiler allocates the 7 bits of a. Then the compiler tries to allocate the 5 bits of b, but this would cross a byte boundary. Because the gap of 1 bit is smaller than the specified gap of 3 bits, b is allocated in the next byte. Then the allocation starts for c. After the allocation of b there are 3 bits left. Because the gap is 3 bits, c is allocated in the next byte. If the maximum gap size were specified to 0, all 16 bits of B would be allocated in two bytes. Since the gap limit is set to 3, and the gap required for allocating c in the next byte is also 3, the compiler will use a 16-bit word as allocation unit. Both b and c will be allocated within this word.

Assuming we initialize an instance of B as below:

```
B s = {2, 7, 5},
```

we get the following memory layouts:

```
-BfaGapLimitBits1 : 53 82

- BfaGapLimitBits3 : 02 00 A7

-BfaGapLimitBits4 : 02 07 05
```

See also

Bitfield Allocation

7.2.3.1.9 -BfaTSR: Bitfield Type Size Reduction

Group

CODE GENERATION

Scope

Function

Syntax

```
-BfaTSR[ON|OFF]
```

Arguments

`ON`: Bit Field Type Size Reduction enabled

OFF: Bit Field Type-Size Reduction disabled

Default

```
HC08: -BfaTSRON
```

Defines

```
__BITFIELD_TYPE_SIZE_REDUCTION__

__BITFIELD_NO_TYPE_SIZE_REDUCTION__
```

Pragmas

None

Description

This option is available whether or not the compiler uses type-size reduction for bitfields. Type-size reduction means that the compiler reduces the type of an `int` bitfield to a `char` bitfield if the `int` bitfield fits into a character. Type-size reduction allows the compiler to allocate memory only for one byte instead of for an integer.

Examples

The following listings demonstrate the effects of `-BfaTSRoff` and `-BfaTSRon`, respectively.

Listing: -BfaTSRoff

```
struct{
  long b1:4;

  long b2:4;

} myBitfield;

31                        7  3  0

---------------------------------

|########################|b2|b1| -BfaTSRoff

---------------------------------
```

Listing: -BfaTSRon

```
7   3    0
----------

|b2 | b1 | -BfaTSRon

----------
```

Example

```
-BfaTSRon
```

See also

Bitfield Type Reduction

7.2.3.1.10 -C++ (-C++f, -C++e, -C++c): C++ Support

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-C++ (f|e|c)
```

Arguments

`f` : Full ANSI Draft C++ support

`e` : Embedded C++ support (EC++)

`c` : compactC++ support (cC++)

Default

None

Defines

```
__cplusplus
```

Pragmas

None

Description

With this option enabled, the Compiler behaves as a C++ Compiler. You can choose between three different types of C++:

- Full ANSI Draft C++ supports the whole C++ language.
- Embedded C++ (EC++) supports a constant subset of the C++ language. EC++ does not support inefficient items like templates, multiple inheritance, virtual base classes and exception handling.
- compactC++ (cC++) supports a configurable subset of the C++ language. You can configure this subset with the option `-Cn`.

If the option is not set, the Compiler behaves as an ANSI-C Compiler.

If the option is enabled and the source file name extension is `*.c`, the Compiler behaves as a C++ Compiler.

If the option is not set, but the source-filename extension is `*.cpp` or `*.cxx`, the Compiler behaves as if the `-C++f` option is set.

Example

```
COMPOPTIONS=-C++f
```

7.2.3.1.11 -Cc: Allocate Const Objects into ROM

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Cc
```

Arguments

None

Default

None

Defines

None

Pragmas

#pragma INTO_ROM: Put Next Variable Definition into ROM

Description

The -Cc command line option puts const objects into the ROM_VAR segment, which the Linker parameter file assigns to a ROM section (refer *Linker* section in the *Build Tool Utilities Reference Manual*).

NOTE

Use this option only for HIWARE object-file formats. In the ELF/DWARF object-file format, the Compiler allocates constants into the .rodata section.

The Linker does not initialize objects allocated into a read-only section. The startup code does not have to copy the constant data.

You may also put variables into the ROM_VAR segment by using the segment pragma (refer *Linker* section in the *Build Tool Utilities* manual). Using the #pragma CONST_SECTION for constant-segment allocation, the Compiler allocates const variables into the ROM_VAR segment.

If the current data segment is not the default segment, const objects remain in the user-defined segment. If the data segment contains *only*const objects, the Compiler may allocate the segment to a ROM memory section (refer to the *Linker* section in the *Build Tool Utilities Reference Manual* for more information).

NOTE

The Compiler uses the default addressing mode for the constants specified by the memory model.

Example

The following listing shows how the -Cc compiler option affects the SECTIONS segment of a PRM file (HIWARE object-file format only).

Listing: Constant Objects into ROM

```
SECTIONS
  MY_ROM  READ_ONLY     0x1000 TO 0x2000

PLACEMENT

  DEFAULT_ROM, ROM_VAR  INTO MY_ROM
```

See also

Segmentation

Linker Manual

-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format option

#pragma INTO_ROM: Put Next Variable Definition into ROM

7.2.3.1.12 -Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-Ccx
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option allows you to use Cosmic style `@near`, `@far` and `@tiny` space modifiers as well as `@interrupt` in your C code. You must switch the `-ANSI` option off before using this option. It is not necessary to remove the Cosmic space modifiers from your application code. You do not need to place the objects in sections addressable by the Cosmic space modifiers.

When the Compiler parses a Cosmic modifier, the following occurs:

- The Compiler always allocates the declared objects in a special Cosmic compatibility (`_cx`...) section, regardless of which section pragmas are set. The target compatibility section used depends on the space modifier, the `const` qualifier, and whether the code

line is a function or a variable. Refer to the following table for object placement information.
- Space modifiers on the left hand side of a pointer declaration specify the pointer type and `pointer size`, depending on the target.

Refer example in the following listing for a PRM file showing section placement for the sections in the following table.

Table 7-6. Cosmic Modifier Handling

Definition	Placement to _CX section
`@tiny int my_var`	`_CX_DATA_TINY`
`@near int my_var`	`_CX_DATA_NEAR`
`@far int my_var`	`_CX_DATA_FAR`
`const @tiny int my_cvar`	`_CX_CONST_TINY`
`const @near int my_cvar`	`_CX_CONST_NEAR`
`const @far int my_cvar`	`_CX_CONST_FAR`
`@tiny void my_fun(void)`	`_CX_CODE_TINY`
`@near void my_fun(void)`	`_CX_CODE_NEAR`
`@far void my_fun(void)`	`_CX_CODE_FAR`
`@interrupt void my_fun(void)`	`_CX_CODE_INTERRUPT`

For further information about porting applications from Cosmic to CodeWarrior IDE refer to technical note TN234 located at <CodeWarrior for Microcontrollers V6.x Installation>\Help\PDF. The following table gives an overview of HC(S)08 space-modifier mapping.

Table 7-7. Cosmic Space Modifier Mapping for HC(S)08

Definition	Keyword Mapping
`@tiny`	`__near`
`@near`	`__far`
`@far`	`__far`

Refer the following listing for an example of the `-Ccx` compiler option.

Listing: Cosmic Space Modifiers

```
volatile @tiny char tiny_ch;
extern @far const int table[100];

static @tiny char * @near ptr_tab[10];

typedef @far int (*@far funptr)(void);

funptr my_fun;  /* banked and __far calling conv. */
```

```
char @tiny *tptr = &tiny_ch;

char @far  *fptr = (char @far *)&tiny_ch;

Example for a prm file:

(16- and 24-bit addressable ROM;

 8-, 16- and 24-bit addressable RAM)

SEGMENTS

  MY_ROM   READ_ONLY     0x2000   TO 0x7FFF;

  MY_BANK  READ_ONLY     0x508000 TO 0x50BFFF;

  MY_ZP    READ_WRITE    0xC0     TO 0xFF;

  MY_RAM   READ_WRITE    0xC000   TO 0xCFFF;

  MY_DBANK READ_WRITE    0x108000 TO 0x10BFFF;

END

PLACEMENT

  DEFAULT_ROM, ROM_VAR,

  _CX_CODE_NEAR, _CX_CODE_TINY, _CX_CONST_TINY,

  _CX_CONST_NEAR               INTO MY_ROM;

  _CX_CODE_FAR, _CX_CONST_FAR  INTO MY_BANK;

  DEFAULT_RAM, _CX_DATA_NEAR   INTO MY_RAM;

  _CX_DATA_FAR                 INTO MY_DBANK;

  _ZEROPAGE, _CX_DATA_TINY     INTO MY_ZP;

END
```

See also

Cosmic Manuals, Build Tools Utilities Reference Manual, TN234

7.2.3.1.13 -Ci: Bigraph and Trigraph Support

Group

LANGUAGE

Scope

Function

Syntax

```
  -Ci
```

Arguments

None

Default

None

Defines

```
__TRIGRAPHS__
```

Pragmas

None

Description

If certain tokens are not available on your keyboard, replace them with the keywords shown in the following table.

Table 7-8. Keyword Alternatives for Unavailable Tokens

Bigraph Keyword	Token Replaced	Trigraph Keyword	Token Replaced	Additional Keyword	Token Replaced
<%	}	??=	#	and	&&
%>	}	??/	\	and_eq	&=
<:	[	??'	^	bitand	&
:>	]	??(	[	bitor	|
%:	#	??)	]	compl	~
%:%:	##	??!	|	not	!
		??<	{	or	||
		??>	}	or_eq	|=
		??-	~	xor	^
				xor_eq	^=
				not_eq	!=

NOTE

Additional keywords are not allowed as identifiers if this option is enabled.

Example

```
-Ci
```

The example in the following listing shows the use of trigraphs, bigraphs and the additional keywords with the corresponding `normal' C source code.

Listing: Trigraphs, Bigraphs, and Additional Keywords

```
int Trigraphs(int argc, char * argv??(??)) ??<
  if (argc<1 ??!??! *argv??(1??)=='??/0') return 0;

  printf("Hello, %s??/n", argv??(1??));

??>

%:define TEST_NEW_THIS 5

%:define cat(a,b) a%:%:b

??=define arraycheck(a,b,c) a??(i??) ??!??! b??(i??)

int i;

int cat(a,b);

char a<:10:>;

char b<:10:>;

void Trigraph2(void) <%

  if (i and ab) <%

    i and_eq TEST_NEW_THIS;

    i = i bitand 0x03;

    i = i bitor 0x8;

    i = compl i;

    i = not i;

  %> else if (ab or i) <%

    i or_eq 0x5;

    i = i xor 0x12;

    i xor_eq 99;

  %> else if (i not_eq 5) <%

    cat(a,b) = 5;

    if (a??(i??) || b[i])<%%>

    if (arraycheck(a,b,i)) <%

      i = 0;

    %>

  %>

%>

/* is the same as ... */

int Trigraphs(int argc, char * argv[]) {
```

```
  if (argc<1 || *argv[1]=='\0') return 0;
  printf("Hello, %s\n", argv[1]);
}
#define TEST_NEW_THIS 5
#define cat(a,b) a##b
#define arraycheck(a,b,c) a[i] || b[i]
int i;
int cat(a,b);
char a[10];
char b[10];
void Trigraph2(void){
  if (i && ab) {
    i &= TEST_NEW_THIS;
    i = i & 0x03;
    i = i | 0x8;
    i = ~i;
    i = !i;
  } else if (ab || i) {
    i |= 0x5;
    i = i ^ 0x12;
    i ^= 99;
  } else if (i != 5) {
    cat(a,b) = 5;
    if (a[i] || b[i]){}
    if (arraycheck(a,b,i)) {
      i = 0;
    }
  }
}
```

7.2.3.1.14 -Cn[={Vf|Tpl|Ptm...}]: Disable compactC++ features

Group

LANGUAGE

Scope

Function

Description

Use this option to disable compactC++ fetaures.

7.2.3.1.15 -Cni: No Integral Promotion on Characters

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Cni
```

Arguments

None

Default

None

Defines

`__CNI__`

Pragmas

None

Description

The `-Cn`i option enhances character operation code density by omitting integral promotion. This option enables behavior that is not ANSI-C compliant.

In ANSI-C operations with data types, anything smaller than `int` must be promoted to `int` (integral promotion). With this rule, adding two unsigned character variables causes the Compiler to zero-extend each character operand, and add them as `int` operands. Integral

promotion is unnecessary when the result will be stored as a character. Using this option, the Compiler avoids promotion when possible. Performing operations on a character base instead of an integer base may decrease code size.

In most expressions, ANSI-C requires `char` type variables to be extended to the next larger type `int`, which requires a 16-bit size, according to ANSI standard. The -Cni option suppresses this ANSI-C behavior and thus allows the use of 'characters' and 'character-sized constants' in expressions.

NOTE

Code generated with this option set does not conform to ANSI standards. Code compiled with this option is not portable. Using this option is not recommended in most cases.

The ANSI standard requires that old-style function declarations using the `char` parameter be extended to `int`. The -Cni option disables this extension and saves additional RAM.

Example

Refer the following listing.

Listing: Definition of an Old Style Function using a char Parameter

```
old_style_func (a, b, c)
  char a, b, c;

{

  ...

}
```

The space reserved for a, b, and c is one byte each, instead of two.

For expressions containing different types of variables, the following conversion rules apply:

- If both variables are of type `signed char`, the expression is evaluated signed.
- If one of two variables is of type `unsigned char`, the expression is evaluated unsigned, regardless of whether the other variable is of type `signed` or `unsigned char`.
- If one operand is of another type than `signed` or `unsigned char`, the usual ANSI-C arithmetic conversions are applied.
- If constants are in the character range, they are treated as characters. Remember that the char type is signed and applies to the constants -128 to 127. All constants greater than 127 are treated as integers. To treat them as characters, cast them as characters (as shown in the following listing).
  Listing: Casting Integers to Signed char

```
signed char a, b;
if (a > b * (signed char)129)
```

NOTE

This option is ignored when the -Ansi Compiler switch is active.

7.2.3.1.16 -Cppc: C++ Comments in ANSI-C

Group

LANGUAGE

Scope

Function

Syntax

```
-Cppc
```

Arguments

None

Default

By default, the Compiler does not allow C++ comments if the -Ansi: Strict ANSI compiler option is set.

Defines

None

Pragmas

None

Description

The -Ansi option forces the compiler to conform to the ANSI-C standard. Because a strict ANSI-C compiler rejects any C++ comments (started with //), use this option to allow C ++ comments (as shown in the following listing).

Listing: Using -Cppc to Allow C++ Comments

```
-Cppc /* This allows the code containing C++ comments to be compiled with

   the
-Ansi option set */

void myfun(void) // this is a C++ comment
```

See also

-Ansi: Strict ANSI compiler option

7.2.3.1.17 -Cq: Propagate const and volatile Qualifiers for Structs

Group

LANGUAGE

Scope

Application

Syntax

```
-Cq
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option propagates `const` and `volatile` qualifiers for structures. That means if all members of a structure are constant, the structure itself is constant as well. The same happens with the volatile qualifier. If the structure is declared as constant or volatile, all members are constant or volatile.

Example

The source code in the following listing declares two `structs`, each of which has a `const` member.

Listing: Declaring Two structs

```
struct {
  const field;
} s1, s2;
void myfun(void) {
  s1 = s2; // struct copy
  s1.field = 3; // error: modifiable lvalue expected
}
```

In the above example, the field in the struct is constant, but not the struct itself. Thus the `struct` copy `` `s1 = s2' `` is legal, even if the field of the struct is constant. But a write access to the `struct` field causes an error message. Using the `-Cq` option propagates the qualification (`const`) of the fields to the whole `struct` or array. In the above example, the `struct` copy causes an error message.

7.2.3.1.18 -C[s08|08]: Generate Code for Specific HC08 Families

Group

CODE GENERATION

Scope

Compilation Unit

Syntax

```
-Cs08
```

Arguments

`s08`: HCS08 family

`08`: HC08 family

Default

`-C08`

Defines

__HCS08__

Pragmas

None

Description

Setting this option generates code for the HC(S)08 family. In addition, using this option allows all new HC(S)08 operation-operand combinations in the HLI.

Example

```
__asm LDHX 2,X;
```

7.2.3.1.19 -CswMaxLF: Maximum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

```
-CswMaxLF<number>
```

Arguments

<number>: a number in the range of 0 to 100 denoting the maximum load factor.

Default

Backend-dependent.

Defines

None

Pragmas

None

Description

Using this option changes the default strategy of the Compiler to use tables for switch statements.

NOTE

This option is only available if the compiler supports switch tables.

Normally the Compiler uses a table for switches with more than about eight labels, provided the table is between 80% (minimum load factor of 80) and 100% (maximum load factor of 100) full. If there are not enough labels for a table or the table is not filled, the Compiler generates a branch tree (tree of if-else-if-else). This branch tree, like an `unrolled' binary search in a table, quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. In addition, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging may be more seamless.

Specifying a load factor means that tables generate in specific `fuel' status:

Listing: Specifying Table Load Factor

```
switch(i) {
  case 0: ...

  case 1: ...

  case 2: ...

  case 3: ...

  case 4: ...

//  case 5: ...

  case 6: ...

  case 7: ...

  case 8: ...

  case 9: ...

  default

}
```

The above table is filled to 90% (labels for `0' to `9', except for `5'). Setting the minimum load factor to 50% and the maximum load factor for the above case to 80%, a branch tree generates instead a table. But setting the maximum load factor to 95% produces a table.

To guarantee that tables generated for full switch tables only, set the table minimum and maximum load factors to 100:

`-CswMinLF100 -CswMaxLF100.`

See also

Compiler options:

- -CswMinLB: Minimum Number of Labels for Switch Tables
- Option -CswMinSLB: Minimum Number of Labels for Switch Search Tables
- -CswMinLF: Minimum Load Factor for Switch Tables

7.2.3.1.20 -CswMinLB: Minimum Number of Labels for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

```
-CswMinLB<number>
```

Arguments

`<number>`: a positive number denoting the number of labels.

Default

Backend-dependent

Defines

None

Pragmas

None

Description

Using this option changes the default strategy of the Compiler to use tables for switch statements.

NOTE

This option is only available if the compiler supports switch tables.

Normally the Compiler uses a table for switches with more than about eight labels. When there are not enough labels for a table, the Compiler generates a branch tree (tree of if-else-if-else). This branch tree, like an `unrolled' binary search in a table, quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table may increase the code execution speed, but probably increases the code size also. In addition, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging may be much easier.

To disable any tables for switch statements, set the minimum number of labels needed for a table to a high value (for example, 9999):

```
-CswMinLB9999 -CswMinSLB9999.
```

When disabling simple tables it usually makes sense also to disable search tables with the `-CswMinSLB` option.

See also

Compiler options :

- -CswMinLF: Minimum Load Factor for Switch Tables
- -CswMinSLB: Minimum Number of Labels for Switch Search Tables
- -CswMaxLF: Maximum Load Factor for Switch Tables

7.2.3.1.21 -CswMinLF: Minimum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

```
-CswMinLF<number>
```

Arguments

`<number>`: a number in the range of 0 - 100 denoting the minimum load factor

Default

Backend-dependent

Defines

None

Pragmas

None

Description

Allows the Compiler to use tables for switch statements.

NOTE

This option is only available if the compiler supports switch tables.

Normally the Compiler uses a table for switches with more than about eight labels, provided the table is between 80% (minimum load factor of 80) and 100% (maximum load factor of 100) full. If there are not enough labels for a table or the table is not full, the Compiler generates a branch tree (tree of if-else-if-else). This branch tree, like an `unrolled' binary search in a table, quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. In addition, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging is more seamless.

Specifying a load factor means that tables generate in specific `fuel' status:

Listing: Specifying Table Load Factor

```
switch(i) {
  case 0: ...

  case 1: ...

  case 2: ...

  case 3: ...

  case 4: ...

  //  case 5: ...

  case 6: ...

  case 7: ...

  case 8: ...

  case 9: ...

  default

}
```

The above table is 90% full (labels for `0' to `9', except for `5'). Setting the maximum load factor to 100% and the minimum load factor for the above case to 90%, this still generates a table. But setting the minimum load factor to 95% produces a branch tree.

To guarantee that tables generate for full switch tables only, set the minimum and maximum table load factors to 100:

```
-CswMinLF100 -CswMaxLF100.
```

See also

Compiler options :

- -CswMinLB: Minimum Number of Labels for Switch Tables
- -CswMinSLB: Minimum Number of Labels for Switch Search Tables
- -CswMaxLF: Maximum Load Factor for Switch Tables

7.2.3.1.22 -CswMinSLB: Minimum Number of Labels for Switch Search Tables

Group

CODE GENERATION

Scope

Function

Syntax

```
-CswMinSLB<number>
```

Arguments

`<number>`: a positive number denoting the number of labels

Default

Backend-dependent

Defines

None

Pragmas

None

Description

Allows the Compiler to use tables for switch statements.

NOTE

This option is only available if the compiler supports search tables.

The Compiler implements switch tables in different ways. When almost all case entries in some range are given, the Compiler uses a table containing only branch targets. Densely populated branch target tables improve efficiency because the Compiler accesses only the correct entry. When large holes exist in some areas, a table form can still be used.

Encoding the case entry and its corresponding branch target into a table creates a search table. Accessing a search table requires a complex runtime routine to check all entries until it finds the matching one. Search tables execute slowly.

Using a search table improves code density, but the execution time increases. Every data request from a search table requires checking every entry until the correct entry is located. For a dense table, the compiler computes the right offset and accesses the table. In addition, note that all backends that implement search tables use a complex runtime routine. This complicates debugging.

To disable search tables for switch statements, set the minimum number of labels needed for a table to a high value (for example, 9999): `-CswMinSLB9999`.

See also

Compiler options :

- -CswMinLB: Minimum Number of Labels for Switch Tables
- -CswMinLF: Minimum Load Factor for Switch Tables
- -CswMaxLF: Maximum Load Factor for Switch Tables

7.2.3.1.23 -Cu: Loop Unrolling

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Cu[=i<number>]
```

Arguments

<number>: number of iterations for unrolling, between 0 and 1024

Default

None

Defines

None

Pragmas

```
#pragma LOOP_UNROLL: Force Loop Unrolling
```

#pragma NO_LOOP_UNROLL: Disable Loop Unrolling

Description

Enables loop unrolling with the following restrictions:

- Only simple `for` statements are unrolled, for example,

  ```
  for (i=0; i<10; i++)
  ```

- Initialization and test of the loop counter must be done with a constant.
- Only <, >, <=, >= are permitted in a condition.
- Only ++ or -- are allowed for the loop variable increment or decrement.
- The loop counter must be integral.
- No change of the loop counter is allowed within the loop.
- The loop counter must not be used on the left side of an assignment.
- No address operator (&) is allowed on the loop counter within the loop.
- Only small loops are unrolled:
  - Loops with few statements within the loop.
  - Loops with fewer than 16 increments or decrements of the loop counter. The bound may be changed with the optional argument `= i<number>`. The `-Cu=i20` option unrolls loops with a maximum of 20 iterations.

Examples

Listing: for Loop

```
-Cu int i, j;
j = 0;
for (i=0; i<3; i++) {
  j += i;
}
```

With the -Cu compiler option given, the Compiler issues an information message '*Unrolling loop*' and transforms this loop as shown in the following listing.

Listing: for Loop

```
j += 1;
j += 2;

i  = 3;
```

The Compiler also transforms some special loops, i.e., loops with a constant condition or loops with only one pass:

Listing: Example for Loop with a Constant Condition

```
for (i=1; i>3; i++) {
  j += i;

}
```

The Compiler issues an information message '*Constant condition found, removing loop*' and transforms the loop into a simple assignment, because the loop body is never executed:

```
i=1;
```

Listing: Example for Loop with Only One Pass

```
for (i=1; i<2; i++) {
  j += i;

}
```

Because the loop body is executed only once, the Compiler issues a warning '*Unrolling loop*' and transforms the for loop into:

```
j += 1;


i  = 2;
```

7.2.3.1.24 -Cx: Switch Off Code Generation

Group

CODE GENERATION

Scope

Compilation Unit

Syntax

```
-Cx
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The `-Cx` Compiler option disables the code generation process of the Compiler. The Compiler generates no object code, although the Compiler performs a syntactical check of the source code. This allows you to check your source code for errors without generating object code.

7.2.3.1.25 -D: Macro Definition

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-D<identifier>[=<value>]
```

Arguments

<identifier>: identifier to be defined

<value>: value for <identifier>, anything except "-" and blank

Default

None

Defines

None

Pragmas

None

Description

The Compiler allows the definition of a macro on the command line. The effect is the same as having a # define directive at the very beginning of the source file.

Listing: DEBUG Macro Definition

```
-DDEBUG=0
This is the same as writing:

#define DEBUG 0

in the source file.
```

To include strings with blanks in your macro definition, either use an escape sequence or use double quotes.

```
-dPath="Path\40with\40spaces"
```

```
-d"Path=""Path with spaces"""
```

NOTE

Blanks are *not* allowed after the -D option; the first blank terminates this option. Also, macro parameters are not supported.

7.2.3.1.26 -Ec: Conversion from 'const T*' to 'T*'

Group

LANGUAGE

Scope

Function

Syntax

```
-Ec
```

Arguments

None

Default

None

Description

Enabling this non-ANSI compliant extension allows the compiler to treat a pointer to a constant type like a pointer to the non-constant equivalent of the type. Earlier Compilers did not check a store to a constant object through a pointer. This option is useful when compiling older source code.

Listing: Converting `const T*' to `T*'

```
void f() {
  int *i;

  const int *j;

  i=j; /* C++ illegal, but with -Ec ok! */

}

struct A {

  int i;

};

void g() {

  const struct A *a;

  a->i=3; /* ANSI C/C++ illegal, but with -Ec ok! */

}

void h() {

  const int *i;

  *i=23; /* ANSI-C/C++ illegal, but with -Ec ok! */
```

```
}
```

Defines

None

Pragmas

None

Listing: Example

```
-Ec
void myfun(const int *p){

  *p = 0; // some Compilers do not issue an error
```

7.2.3.1.27 -Eencrypt: Encrypt Files

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Eencrypt[=<filename>]
```

Arguments

`<filename>`: The name of the file to be generated

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

Default

The default filename is %f.e%e. A file named ` `myfun.c`' creates an encrypted file named ` `myfun.ec`'.

Description

This option encrypts all files passed together with this option, using the given key with the `-Ekey` option (refer -Ekey: Encryption Key).

NOTE

This option is only available or operative with a license for the following feature: `HIxxxx30`, where `xxxx` is the feature number of the compiler for a specific target.

Defines

None

Pragmas

None

Example

```
myfun.c myfun.h -Ekey1234567 -Eencrypt=%n.e%e
```

This encrypts the ` `myfun.c`' file using the `1234567` key to the ` `myfun.ec`' file and the ` `myfun.h`' file to the ` `myfun.eh`' file.

The encrypted `myfun.ec` and `myfun.eh` files may be passed to a client. The client can compile the encrypted files without the key by compiling the following file:

```
myfun.ec
```

See also

-Ekey: Encryption Key

7.2.3.1.28 -Ekey: Encryption Key

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Ekey<keyNumber>
```

Arguments

```
<keyNumber>
```

Default

The default encryption key is `0'. Using this default is not recommended.

Description

Use this option to encrypt files with the given key number (`-Eencrypt` option).

NOTE

This option is only available or operative with a license for the following feature: `HIxxxx30`, where `xxxx` is the feature number of the compiler for a specific target.

Defines

None

Pragmas

None

Example

```
myfun.c -Ekey1234567 -Eencrypt=%n.e%e
```

This encrypts the ` `myfun.c`' file using the `1234567` key.

See also

-Eencrypt: Encrypt Files

7.2.3.1.29 -Env: Set Environment Variable

Group

HOST

Scope

Compilation Unit

Syntax

```
-Env<Environment Variable>=<Variable Setting>
```

Arguments

`<Environment Variable>`: Environment variable to be set

`<Variable Setting>`: Setting of the environment variable

Default

None

Description

This option sets an environment variable. Use this environment variable in the maker, or use to overwrite system environment variables.

Defines

None

Pragmas

None

Example

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

This is the same as:

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

in the `default.env` file.

Use the following syntax to use an environment variable that uses filenames with spaces:

```
-Env"OBJPATH=\program files"
```

See also

Environment

7.2.3.1.30 -F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format

Group

OUTPUT

Scope

Application

Syntax

```
-F(1|1o|2|2o|6|7|h)
```

Arguments

`h`: HIWARE object-file format

`1`: ELF/DWARF 1.1 object-file format

`1o`: compatible ELF/DWARF 1.1 object-file format

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

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

`6`: strict HIWARE V2.6 object-file format

`7`: strict HIWARE V2.7 object-file format

NOTE

Not all object-file formats may be available for a target.

Default

```
-Fh
```

Defines

```
__HIWARE_OBJECT_FILE_FORMAT__
```

```
__ELF_OBJECT_FILE_FORMAT__
```

Pragmas

None

Description

The Compiler writes the code and debugging info into an object file after compilation.

The Compiler uses a HIWARE-proprietary object-file format when the `-Fh`, `-F6` or `-F7` options are set.

The HIWARE Object-file Format (`-Fh`) has the following limitations:

- The type char is limited to a size of 1 byte.

- Symbolic debugging for enumerations is limited to 16-bit signed enumerations.
- No zero bytes in strings are allowed (a zero byte marks the end of the string).

The HIWARE V2.7 Object-file Format (option `-F7`) has some limitations:

- The type char is limited to a size of 1 byte.
- Enumerations are limited to a size of 2 bytes and must be signed.
- No symbolic debugging for enumerations.
- The standard type short is encoded as int in the object-file format.
- No zero bytes in strings allowed (a zero byte marks the end of the string).

The Compiler produces an ELF/DWARF object file when the `-F1` or `-F2` options are set. This object-file format may also be supported by other Compiler vendors.

In the Compiler ELF/DWARF 2.0 output, some constructs written in previous versions did not conform to the ELF standard because the standard was not clear enough in this area. Because old versions of the simulator or debugger (V5.2 or earlier) are not able to load the corrected format, the old behavior can still be produced by using `-f2o` instead of `-f2`. Some old versions of the debugger (simulator or debugger V5.2 or earlier) generate a GPF when a new absolute file is loaded. To use the older versions, use `-f2o` instead of `-f2`. New versions of the debugger can load both formats correctly. Also, some older ELF/DWARF object file loaders from emulator vendors may require you to set the `-F2o` option.

The `-F1o` option is only supported if the target supports the ELF/DWARF 1.1 format. This option is only used with older debugger versions as a compatibility option. This option may be discontinued in the future. We recommend that you use `-F1` instead.

We recommend that you use the ELF/DWARF 2.0 format instead of the ELF/DWARF 1.1. The 2.0 format is much more generic. In addition, it supports multiple include files plus modifications of the basic generic types (for example, floating point format). Debug information is also more robust.

7.2.3.1.31 -Fd: Double is IEEE32

Group

CODE GENERATION

Scope

Application

Syntax

```
-Fd
```

Arguments

None

Default

None

Defines

Refer -T: Flexible Type Management

Pragmas

See -T

Description

Allows you to change the float or double format. By default, float is `IEEE32` and doubles are `IEEE64`.

When you set this option, all doubles are in `IEEE32` instead of `IEEE64`.

Floating point formats may be also changed with the `-T` option.

7.2.3.1.32 -H: Short Help

Group

VARIOUS

Scope

None

Syntax

```
-H
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -H option causes the Compiler to display a short help list of available options within the Compiler window. Options are grouped into HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGES, and VARIOUS.

Do not specify any other option or source file when the invoking -H option.

Example

The following listing shows the short list options.

Listing: Short Help Options

```
-H may produce the following list:
INPUT:

-!      Filenames are clipped to DOS length

-I      Include file path

VARIOUS:

-H      Prints this list of options

-V      Prints the Compiler version
```

7.2.3.1.33 -I: Include File Path

Group

INPUT

Scope

Compilation Unit

Syntax

```
-I<path>
```

Arguments

<path>: path, terminated by a space or end-of-line

Default

None

Defines

None

Pragmas

None

Description

Allows you to set include paths in addition to the LIBPATH, LIBRARYPATH, and GENPATH environment variables (refer LIBRARYPATH: `include <File>' Path and GENPATH: #include "File" Path). Paths specified with this option have precedence over includes in the current directory, and over paths specified in GENPATH, LIBPATH, and LIBRARYPATH.

Example

```
-I. -I..\h -I\src\include
```

This directs the Compiler to search for header files first in the current directory (.), then relative from the current directory in '..\h', and then in '\src\include'. If the file is not found, the search continues with GENPATH, LIBPATH and LIBRARYPATH for header files in double quotes (#include"headerfile.h"), and with LIBPATH and LIBRARYPATH for header files in angular brackets (#include <stdio.h>).

See also

Input Files

-AddIncl: Additional Include File

LIBRARYPATH: `include <File>' Path

GENPATH: #include "File" Path

7.2.3.1.34 -La: Generate Assembler Include File

Group

OUTPUT

Scope

Function

Syntax

```
-La[=<filename>]
```

Arguments

`<filename>`: The name of the file to be generated

It may contain special modifiers (refer Using Special Modifiers)

Default

No file created

Defines

None

Pragmas

None

Description

The `-La` option causes the Compiler to generate an assembler include file when the `CREATE_ASM_LISTING` pragma occurs. This option specifies the name of the created file. If no name is specified, the compiler takes a default of `%f.inc`. To put the file into the directory specified by the TEXTPATH: Text File Path environment variable, use the option `-la=%n.inc`. The `%f` option already contains the path of the source file. When you use `%f`, the compiler puts the generated file in the same directory as the source file.

The content of all modifiers refers to the main input file and not to the actual header file. The main input file is the one specified on the command line.

Example

```
-La=asm.inc
```

See also

#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing

TEXTPATH: Text File Path

7.2.3.1.35 -Lasm: Generate Listing File

Group

OUTPUT

Scope

Function

Syntax

```
-Lasm[=<filename>]
```

Arguments

`<filename>`: The name of the file to be generated.

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

Default

No file created.

Defines

None

Pragmas

None

Description

The `-Lasm` option causes the Compiler to generate an assembler listing file directly. The Compiler also prints all assembler-generated instructions to this file. The option specifies the name of the file. If you do not specify a name, the Compiler takes a default of `%n.lst`. If the resulting filename contains no path information the Compiler uses the TEXTPATH environment variable (refer TEXTPATH: Text File Path).

The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, use this option only to review the generated code. It cannot currently be used to generate a file for assembly.

Example

```
-Lasm=asm.lst
```

See also

-Lasmc: Configure Listing File

7.2.3.1.36 -Lasmc: Configure Listing File

Group

OUTPUT

Scope

Function

Syntax

```
-Lasmc[={a|c|i|s|h|p|e|v|y}]
```

Arguments

a: Do not write the address in front of every instruction

c: Do not write the hex bytes of the instructions

i: Do not write the decoded instructions

s: Do not write the source code

h: Do not write the function header

p: Do not write the source prolog

e: Do not write the source epilog

v: Do not write the compiler version

y: Do not write cycle information

Default

All printed together with the source

Defines

None

Pragmas

None

Description

The `-Lasmc` option configures the output format of the listing file generated with the -Lasm: Generate Listing File option. The addresses, the hex bytes, and the instructions are selectively switched off.

The format of the listing file uses the layout shown in the following listing. The letters in brackets ([]) indicate which suboption to use to switch the instruction off.

Listing: -Lasm Configuration Options

```
[v] ANSI-C/cC++ Compiler V-5.0.1
[v]

[p]    1:

[p]    2:  void myfun(void) {

[h]

[h] Function: myfun

[h] Source  : C:\Freescale\test.c

[h] Options : -Lasm=%n.lst

[h]

[s]    3:  }

[a]  0000 [c] 3d          [i]  RTS

[e]    4:

[e]    5:  // comments

[e]    6:
```

Example

```
-Lasmc=ac
```

7.2.3.1.37 -Ldf: Log Predefined Defines to File

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Ldf[=<file>]
```

Arguments

<file>: filename for the log file, default is predef.h.

Default

Default <file> is predef.h.

Defines

None

Pragmas

None

Description

The -Ldf option causes the Compiler to generate a text file that contains a list of the compiler-defined #define. The default filename of predef.h may be changed (for example, -Ldf="myfile.h"). The Compiler generates the file in the directory specified by the TEXTPATH environment variable (refer TEXTPATH: Text File Path). The defines written to this file depend on the actual Compiler option settings (such as type size settings or ANSI compliance).

NOTE

The defines specified by the command line (-D: Macro Definition option) are not included.

This option may be very useful for SQA. With this option it is possible to document every #define used to compile all sources.

NOTE

This option only has an effect when a file is compiled, otherwise it has no meaning.

Example

The following listing lists the contents of a file containing define directives.

Listing: Displays the Contents of a File Containing Define Directives

```
-Ldf This generates the `predef.h' filewith the following content:
/* resolved by preprocessor: __LINE__ */
/* resolved by preprocessor: __FILE__ */
/* resolved by preprocessor: __DATE__ */
/* resolved by preprocessor: __TIME__ */
#define __STDC__ 0
```

```
#define __VERSION__ 5004
#define __VERSION_STR__ "V-5.0.4"
#define __SMALL__
#define __PTR_SIZE_2__
#define __BITFIELD_LSBIT_FIRST__
#define __BITFIELD_MSBYTE_FIRST__
...
```

See also

-D: Macro Definition

7.2.3.1.38 -Li: List of Included Files to ".inc" File

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Li
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -Li option causes the Compiler to generate a text file which contains a list of the #include files specified in the source. This text file shares the same name as the source file but with the extension * .inc. The Compiler stores the file in the path specified by the TEXTPATH: Text File Path environment variable. The generated file may be used in make files.

Example

The following listing uses the -Li compiler option to display a file's contents when that file contains an included directive.

Listing: Display Contents of a File Containing Include Directives

```
-Li If the source file is: `C:\myFiles\b.c':
/* C:\myFiles\b.c */
#include <string.h>
Then the generated file is:
C:\myFiles\b.c :\
C:\Freescale\lib\targetc\include\string.h \
C:\Freescale\lib\targetc\include\libdefs.h \
C:\Freescale\lib\targetc\include\hidef.h \
C:\Freescale\lib\targetc\include\stddef.h \
C:\Freescale\lib\targetc\include\stdtypes.h
```

See also

-Lm: List of Included Files in Make Format compiler option

7.2.3.1.39 -Lic: License Information

Group

VARIOUS

Scope

None

Syntax

```
-Lic
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The `-Lic` option prints the current license information. This information also appears in the *About* window.

Example

```
-Lic
```

See also

Compiler options :

-LicA: License Information about Every Feature in Directory

-LicBorrow: Borrow License Feature

-LicWait: Wait until Floating License is Available from Floating Li- cense Server

7.2.3.1.40 -LicA: License Information about Every Feature in Directory

Group

VARIOUS

Scope

None

Syntax

```
-LicA
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The `-LicA` option prints the license information about every tool or `*.dll` in the same directory with the executable file. This takes some time as the option analyzes every file in the directory.

Example

```
-LicA
```

See also

Compiler options :

-Lic: License Information

-LicBorrow: Borrow License Feature

-LicWait: Wait until Floating License is Available from Floating Li- cense Server

7.2.3.1.41 -LicBorrow: Borrow License Feature

Group

HOST

Scope

None

Syntax

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

Arguments

`<feature>`: the feature name to be borrowed (for example, `HI100100`).

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

<date>: date with optional time specifying when the feature must be returned (for example, 15-Mar-2005:18:35).

Default

None

Defines

None

Pragmas

None

Description

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

Specify the feature name and the date you will return the feature. If the feature you want to borrow belongs to the tool with which you use this option, then you do not need to specify the feature version. However, to borrow any other feature, specify the feature version as well.

Check the status of currently borrowed features in the tool *About* window.

NOTE

You can only borrow features if you have a floating license enabled for borrowing. Refer *FLEXlm* documentation for details on borrowing.

Example

```
-LicBorrowHI100100;3.000:12-Mar-2006:18:25
```

See also

Compiler options:

-LicA: License Information about Every Feature in Directory

-Lic: License Information

-LicWait: Wait until Floating License is Available from Floating Li- cense Server

7.2.3.1.42 -LicWait: Wait until Floating License is Available from Floating Li- cense Server

Group

HOST

Scope

None

Syntax

```
-LicWait
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

By default, if a license is not available from the floating license server, then the application returns immediately. With `-LicWait` set, the application waits (blocking) until a license is available from the floating license server.

Example

```
-LicWait
```

See also

Compiler options:

-Lic: License Information

-LicA: License Information about Every Feature in Directory

-LicBorrow: Borrow License Feature

7.2.3.1.43 -Ll: Write Statistics Output to File

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Ll[=<filename>]
```

Arguments

`<filename>`: file to be used for the output

Default

The default output filename is `logfile.txt`

Defines

None

Pragmas

None

Description

Using the `-Ll` option, the Compiler appends statistical information about the compilation session to the specified file. The information includes Compiler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) for each procedure of the compiled file. The Compiler appends the information to the specified filename (or the file `make.txt`, if no argument given). Set the `TEXTPATH` environment variable (refer TEXTPATH: Text File Path) to store the file into the path specified by the environment variable. Otherwise the Compiler stores the file in the current directory.

Example

The following listing uses the `-Ll` compiler options to add statistical information to the end of an output listing file.

Listing: Statistical Information Appended to Assembler Listing

```
-Ll=mylog.txt
/* myfun.c */

int Func1(int b) {

  int a = b+3;

  return a+2;

}

void Func2(void) {

}

Appends the following two lines into mylog.txt:

myfun.c Func1 -Ll=mylog.txt   11  4 0.055000

myfun.c Func2 -Ll=mylog.txt       1  0 0.001000
```

7.2.3.1.44 -Lm: List of Included Files in Make Format

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Lm[=<filename>]
```

Arguments

`<filename>`: file to be used for the output

Default

The default filename is `Make.txt`

Defines

None

Pragmas

None

Description

Using the -Lm option, the Compiler generates a text file containing a list of the #include files specified in the source. Use the -Lm option when creating make files. The output from several source files may be copied and grouped into one make file. The generated list is in the make format. The filename does not include the path. After each entry, an empty line is added. The Compiler appends the information to the specified filename (or the file make.txt, if no argument is given). Set the TEXTPATH environment variable (refer TEXTPATH: Text File Path) to store the file into the path specified by the environment variable. Otherwise the Compiler stores the file in the current directory.

Example

The example in the following listing shows that the Compiler uses the -Lm option to generate a make file containing include directives.

Listing: Make File Construction

```
COMPOTIONS=-Lm=mymake.txt
Compiling the following sources 'myfun.c' and 'second.c':

/* myfun.c */

#include <stddef.h>

#include "myheader.h"

...

/* second.c */

#include "inc.h"

#include "header.h"

...

This adds the following entries in the 'mymake.txt':

myfun.o :     myfun.c stddef.h myheader.h

seconde.o : second.c inc.h header.h
```

See also

-Li: List of Included Files to ".inc" File

-Lo: Append Object File Name to List (enter [<files>])

Make Utility

7.2.3.1.45 -LmCfg: Configuration for List of Included Files in Make Format (option -Lm)

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-LmCfg[={i|l|m|o|u}]
```

Arguments

i: Write path of included files

l: Use line continuation

m: Write path of main file

o: Write path of object file

u: Update information

x: Unix style paths

Default

None

Defines

None

Pragmas

None

Description

Use this option when configuring the `-Lm` option (refer -Lm: List of Included Files in Make Format). `-LmCfg` operates only when also using the `-Lm` option. The `-Lm` option produces the `dependency' information for a make file. Each dependency information grouping is structured as shown:

```
<main object file>: <main source file> {<included file>}
```

Example

Compile a file named `b.c` which includes `stdio.h`. The `-Lm` option produces the following:

```
b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption, `-Ml', uses line continuation for each single entry in the dependency list. This improves readability as shown in the following listing.

Listing: Using Line Continuations for Clarity

```
b.o: \
  b.c \

  stdio.h \

  stddef.h \

  stdarg.h \

  string.h
```

Use the suboption `-Mm' to write the full path of the compiled file. This is necessary when there are files with the same name in different directories:

```
b.o: C:\test\b.c stdio.h stddef.h stdarg.h string.h
```

The suboption `-Mo' writes the full name of the target object file:

```
C:\test\obj\b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption `-Mi' writes the full path of all included files in the dependency list:

```
b.o: b.c C:\Freescale\lib\include\stdio.h
C:\Freescale\lib\include\stddef.h
C:\Freescale\lib\include\stdarg.h
C:\Freescale\lib\include\
C:\Freescale\lib\include\string.h
```

The suboption `-Mu' updates the information in the output file. If the file does not exist, the Compiler creates the file. If the file exists and the current information is not yet in the file, the Compiler appends the information to the file. If the information is already present, the Compiler updates the information. This allows you to specify this suboption for each compilation ensuring that the make dependency file is always up to date.

Example

```
COMPOPTIONS=-LmCfg=u
```

See also

-Li: List of Included Files to ".inc" File

-Lo: Append Object File Name to List (enter [<files>])

-Lm: List of Included Files in Make Format

Make Utility

7.2.3.1.46 -Lo: Append Object File Name to List (enter [<files>])

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Lo[=<filename>]
```

Arguments

`<filename>`: file to be used for the output

Default

The default filename is `objlist.txt`

Defines

None

Pragmas

None

Description

Using the `-Lo` option, the Compiler appends the object filename to the list in the specified file. The Compiler appends the information to the specified filename (or the file `make.txt`, if no argument given). Set the `TEXTPATH` environment variable (refer TEXTPATH: Text File Path) to store the file into the path specified by the environment variable. Otherwise, the Compiler stores the file in the current directory.

See also

-Li: List of Included Files to ".inc" File

-Lm: List of Included Files in Make Format

7.2.3.1.47 -Lp: Preprocessor Output

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-Lp[=<filename>]
```

Arguments

`<filename>`: The name of the file to be generated.

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

Default

No file created

Defines

None

Pragmas

None

Description

Using the `-Lp` option, the Compiler generates a text file containing the preprocessor's output. If no filename is specified, the text file shares the same name as the source file but with the extension, * `.PRE` (`%n.pre`). The Compiler uses the `TEXTPATH` environment variable to store the preprocessor file.

The resultant file is a form of the source file, with all preprocessor commands (such as `#include`, `#define`, and `#ifdef`) resolved. Only source code is listed with line numbers.

See also

-LpX: Stop after Preprocessor

7.2.3.1.48 -LpCfg: Preprocessor Output Configuration

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-LpCfg[={c|e|f|l|m|n|q|s}]
```

Arguments

`s`: Reconstruct spaces

`q`: Handle single quote ['] as normal token

`n`: No string concatenation

`m`: Do not emit file names

`l`: Emit #line directives in preprocessor output

`f`: Filenames with path

`e`: Emit empty lines

`c`: Do not emit line comments

NOTE

It is necessary to use `q` option when invoking the preprocessor on linker parameter files (`.prm`), because such files may contain linear address specifiers, for example `0x014000'F`.

Default

If `-LpCfg` is specified, all suboptions (arguments) are enabled

Defines

None

Pragmas

None

Description

The `-LpCfg` option specifies source file and `-line` information formatting in the preprocessor output. Switching `-LpCfg` off formats the output as in former compiler versions. Refer the following table for a list of argument effects.

Table 7-9. Effects of Source and Line Information Format Control Arguments

Argument	On	Off
c	typedef unsigned int size_t ;typedef signed int ptrdiff_t ;	/* 22 */ typedef unsignedint size_t ;/* 35 */ typedef signedint ptrdiff_t ;
e	int j;int i;	int j;int i;
f	/**** FILE '<CWInstallDir>\MCU \lib\hc08c\include\hidef.h' */	/**** FILE 'hidef.h' */
l	#line 1 "hidef.h"	/**** FILE 'hidef.h' */
n	/* 9 */ foo ("abc" "def") ;	/* 9 */ foo ("abcdef") ;
m		/**** FILE 'hidef.h' */
s	/* 22 */ typedefunsigned int size_t;/* 35 */ typedef signedint ptrdiff_t;/* 44 */ typedefunsigned char wchar_t;	/* 22 */ typedef unsignedint size_t ;/* 35 */ typedef signedint ptrdiff_t ;/* 44 */ typedef unsignedchar wchar_t ;
all	#line 1 "<CWInstallDir>\MCU \lib\hc08c\include\hidef.h"	/**** FILE 'hidef.h' *//* 20 */

NOTE

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

Example

```
-Lpcfg
```

```
-Lpcfg=lfs
```

See also

-Lp: Preprocessor Output

7.2.3.1.49 -LpX: Stop after Preprocessor

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-LpX
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Without this option, the compiler always translates the preprocessor output as C code. To do only preprocessing, use this option together with the `-Lp` option. No object file is generated.

Example

```
-LpX
```

See also

-Lp: Preprocessor Output

7.2.3.1.50 -M (-Mb, -Ms, -Mt): Memory Model

Group

CODE GENERATION

Scope

Application

Syntax

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

Arguments

`b`: banked memory model

`s`: small memory model

`t`: tiny memory model

Default

```
-Ms
```

Defines

```
__BANKED__
```

```
__SMALL__
```

```
__TINY__
```

Pragmas

#pragma CODE_SEG: Code Segment Definition

Description

The Compiler defaults to the small memory model. The small memory model assumes that all pointers and functions have 16-bit addresses, and requires code and data to be located in the 64-kilobyte address space. Specifying the small memory model when `-MMU` is enabled causes the Compiler to consider any `__far` function accessible only through the code banking mechanism (i.e., using a `CALL` instruction). The tiny memory model assumes that data pointers have 8-bit addresses unless explicitly specified with the keyword `__far`. The banked memory model is available only when `-MMU` is enabled. Memory banking allows you to place program code into at most 256 pages of 16 kilobytes each, but does not affect data allocation. When using banked mode, the compiler considers all functions paged functions unless otherwise specified. Refer Memory Models.

7.2.3.1.51 -MMU: Enable Memory Management Unit (MMU) Support

Group

CODE GENERATION

Scope

Compilation Unit

Syntax

```
-MMU
```

Arguments

None

Default

None

Defines

```
__MMU__
```

Pragmas

None

Description

Setting this option informs the compiler that `CALL` and `RTC` instructions are available, enabling code banking, and that the current architecture has extended data access capabilities, enabling support for `__linear` data types. This option can be used only when `-Cs08` is enabled.

Example

```
__asm CALL myfun;
```

7.2.3.1.52 -N: Show Notification Box in Case of Errors

Group

MESSAGES

Scope

Function

Syntax

```
-N
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Using this option makes the Compiler display an Alert if an error occurs during compilation. This is useful when running a make file (refer *Make Utility*) because the Compiler waits for you to acknowledge the message, thus suspending make file processing.

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

Example

```
-N
```

If an error occurs during compilation, a dialog similar to the following figure appears.

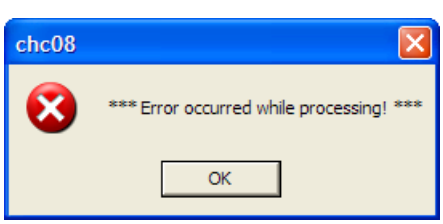

Figure 7-3. Alert Dialog

7.2.3.1.53 -NoBeep: No Beep in Case of an Error

Group

MESSAGES

Scope

Function

Syntax

```
-NoBeep
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Use this option to implement a `beep' notification at the end of processing if an error occurs. To implement a silent error, use this option to switch the beep off.

Example

```
-NoBeep
```

7.2.3.1.54 -NoClrVol: Do not use CLR for volatile variables in the direct page

Group

CODE GENERATION

Syntax

-NoClrVol

Arguments

None

Description

Inhibits the use of CLR for volatile variables in the direct page. The CLR instruction on HC08 has a read cycle. This may lead to unwanted lateral effects (e.g. if the variable is mapped over a hardware register).

7.2.3.1.55 -NoDebugInfo: Do not Generate Debug Information

Group

OUTPUT

Scope

None

Syntax

```
-NoDebugInfo
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The compiler generates debug information by default. Use this option to prevent generation of debug information.

NOTE

To generate an application without debug information in ELF, the linker provides an option to strip the debug information. By calling the linker twice, you can generate two versions of the application: one with and one without debug information. Use this compiler option only if object files or libraries are to be distributed without debug info.

NOTE

This option does not affect the generated code. Only the debug information is excluded.

See also

Compiler options:

-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format

-NoPath: Strip Path Info

7.2.3.1.56 -NoEnv: Do Not Use Environment

Group

STARTUP. This option cannot be specified interactively.

Scope

None

Syntax

```
-NoEnv
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

You can only specify this option at the command line while starting the application. It cannot be specified in any other way, including via the `default.env` file, the command line, or processes.

When you use this option, the application has no environment (`default.env`, `project.ini`, or tips file) data.

Example

```
<compiler>.exe -NoEnv
```

Use the compiler executable file name instead of `<compiler>`.

See also

Local Configuration File

7.2.3.1.57 -NoPath: Strip Path Info

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-NoPath
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Using this option, you can avoid any path information in object files. This is useful if you want to move object files to another file location, or hide your path structure.

See also

-NoDebugInfo: Do not Generate Debug Information

7.2.3.1.58 -O(-Os, -Ot): Main Optimization Target

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-O(s|t)
```

Arguments

`s`: Optimize for code size (default)

`t`: Optimize for execution speed

Default

```
-Os
```

Defines

```
__OPTIMIZE_FOR_SIZE__
```

```
__OPTIMIZE_FOR_TIME__
```

Pragmas

None

Description

At times the Compiler must choose between generating fast, large code, or slow, small code.

The Compiler generally optimizes on code size. It often decides between a runtime routine or expanded code. The programmer can decide whether to choose the slower and shorter or the faster and longer code sequence by setting a command line switch.

The `-Os` option directs the Compiler to optimize the code for smaller code size. The Compiler trades faster/larger code for slower/smaller code.

The `-Ot` option directs the Compiler to optimize the code for faster execution time. The Compiler trades slower/smaller code for faster/larger code.

NOTE

This option only affects some special code sequences. Set this option together with other optimization options (such as register optimization) to get best results.

Example

```
-Os
```

7.2.3.1.59 -O0 : Disable Optimizations

Group

OPTIMIZATIONS

Scope

Function

Description

Use this option to disable all optimizations.

7.2.3.1.60 -Obfv: Optimize Bitfields and Volatile Bitfields

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Obfv
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Use this option to optimize bitfields and volatile bitfields. The compiler changes the access order or combines many accesses into one, even if the bitfields are declared as volatile.

Example

The following listing contains bitfields to be optimized with the `-Obfv` compiler option.

Listing: Bitfields Example

```
volatile struct {
  unsigned int b0:1;

  unsigned int b1:1;

  unsigned int b2:1;

} bf;

void myfun(void) {

  bf.b0 = 1;  bf.b1 = 1;  bf.b2 = 1;

}
```

Using the `-Obfv` option, bitfield access looks like:

```
BSET  bf,#7
```

Without using the `-Obfv` option, bitfield access looks like:

```
BSET  bf,#1


BSET  bf,#2


BSET  bf,#4
```

7.2.3.1.61 -ObjN: Object File Name Specification

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-ObjN=<file>
```

Arguments

`<file>`: Object filename

Default

```
-ObjN=%(OBJPATH)\%n.o
```

Defines

None

Pragmas

None

Description

The object file uses the same name as the processed source file, but with the extension "*.o". This option allows flexible object filename definition. It may contain special modifiers (refer Using Special Modifiers). When <file> in the option contains a path (absolute or relative), the Compiler ignores the OBJPATH environment variable.

Example

```
-ObjN=a.out
```

This example names the resulting object file a.out. Setting the OBJPATH environment variable to \src\obj results in an object file called \src\obj\a.out.

```
fibo.c -ObjN=%n.obj
```

This results in an object file called fibo.obj.

```
myfile.c -ObjN=..\objects\_%n.obj
```

The Compiler names the object file relative to the current directory, which results in an object file called ..\objects_myfile.obj. The Compiler ignores the OBJPATH environment variable because the <file> contains a path.

See also

OBJPATH: Object File Path environment variable

7.2.3.1.62 -Oc: Common Subexpression Elimination (CSE)

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Oc
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option is disabled and present only for compatibility reasons for the HC(S)08.

Performs common subexpression elimination (CSE). The Compiler generates the code for common subexpressions and assignments only once and reuses the result. Depending on available registers, a common subexpression may produce more code due to multiple spills.

NOTE

When the CSE is switched on, variable changes by aliases may generate incorrect optimizations.

Example

```
-Oc
```

In the following listing using the CSE option causes incorrect optimizations. This option is disabled for the HC(S)08.

Listing: Example of Using CSE and Producing Incorrect Results

```
void main(void) {
  int x;

  int *p;

  x = 7;  /* here the value of x is set to 7 */

  p = &x;

  *p = 6;  /* here x is set to 6 by the alias *p */

  /* here x is assumed to be equal to 7 and

     Error is called */

  if(x != 6) Error();
```

```
}
```

NOTE

This error does not occur if you declare `x` as volatile.

7.2.3.1.63 -OdocF: Dynamic Option Configuration for Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OdocF="<option>"
```

Arguments

<option>: Set of options, separated by `'|'` to be evaluated for each single function.

Default

None

Defines

None

Pragmas

None

Description

Normally, you must set a specific set of Compiler switches for each compilation unit (file to be compiled). For some files, a specific set of options may decrease the code size, but for other files, the same set of Compiler options may produce more code depending on the sources.

Some optimizations may reduce the code size for some functions, but may increase the code size for other functions in the same compilation unit. Normally it is impossible to vary options over different functions, or to find the best combination of options.

This option allows the Compiler to choose from a set of options to reach the smallest code size for every function. Without this feature, you must set fixed Compiler switches over the whole compilation unit. With this feature, the Compiler finds the best option combination from a user-defined set for every function.

Standard merging rules apply for this option:

```
-Or -OdocF="-Ocu|-Cu"
```

This is the same as:

```
-OrDOCF="-Ouc|-Cu"
```

The Compiler attempts to find the best combination of the specified options and evaluates all possible combinations of all specified sets, for example, for the option shown in the followingthe following listing.

Listing: Example of Dynamic Option Configuration

```
-W2 -OdocF="-Or|-Cni -Cu|-Oc"
```

The Compiler evaluates code sizes for the following option combinations:

```
1. -W2

2. -W2  -Or

3. -W2      -Cni -Cu

4. -W2  -Or -Cni -Cu

5. -W2               -Oc

6. -W2  -Or          -Oc

7. -W2      -Cni -Cu     -Oc

8. -W2  -Or -Cni -Cu       -Oc
```

The Compiler evaluates all combinations. Thus, specifying more sets takes more evaluation time (for example, for five sets 32 evaluations).

NOTE

Do not specify options with scope Application or Compilation Unit (as memory model, float or double format, or object-file format) or options for the whole compilation unit (like inlining or macro definition) in this option. The generated functions may be incompatible for linking and executing.

Limitations :

- The maximum set of options set is limited to five, for example,

```
-OdocF="-Or -Ou|-Cni|-Cu|-Oic2|-W2 -Ob"
```

- The maximum length of the option is 64 characters.
- The feature is available only for functions and function-compatible options.

Example

```
-Odocf="-Or|-Cni"
```

7.2.3.1.64 -Of and-Onf: Create Sub-Functions with Common Code

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-O[nf|f]
```

Arguments

`nf`: Disable

`f`: Enable

Default

`-Of` default or with `-Os`, `-Onf` with `-Ot`

Defines

None

Pragmas

None

Description

This option performs the reverse of inlining. It detects common code parts in the generated code. The Compiler moves the common code to a different place and replaces all occurrences with a JSR to the moved code. At the end of the common code, the Compiler inserts an RTS instruction. The Compiler increases all SP uses by an address size. This optimization takes care of stack allocation, control flow, and functions having arguments on the stack.

NOTE

Inline assembler code is never treated as common code.

Example

Consider this function:

```
void f(int);

void g(void);

void h(void);

void main(void) {

  f(1);  f(2);  f(3);

  h();

  f(1);  f(2);   f(3);
```

```
    g();

    f(1);  f(2);

  }
```

The compiler first detects that `"f(1); f(2); f(3);"` occurs twice and places this code separately. Then the Compiler replaces the two code patterns by a call to the moved code.

The non-C pseudo code (C does not support local functions) in the following code presents a similar scenario:

```
void main(void) {

  void tmp0(void) {

    f(1);  f(2);  f(3);

  }

  tmp0();

  h();

  tmp0();

  g();

  f(1);  f(2);
```

```
}
```

In a next step, the compiler detects that the code `"f(1); f(2);"` occurs twice. The Compiler generates a second local function:

```
void main(void) {

  void tmp1(void) {

    f(1);  f(2);

  }

  void tmp0(void) {

    tmp1();  f(3);

  }

  tmp0();

  h();

  tmp0();

  g();

  tmp1();
```

```
}
```

`main ()` calls `tmp1` once directly and once indirectly using `tmp0`. These two call chains use a different amount of stack. Because of this situation, it is not always possible to generate correct debug information. For the local function `tmp1`, the compiler cannot state both possible SP values, and therefore only states one of them. While debugging the other state, the debugger declares local variables and the call chain invalid.

Tip

Switch off this optimization to debug your application. The common code complicates the control flow. Also, the debugger cannot distinguish between two distinct usages of the same code. Setting a breakpoint in common code stops the application and every use of it. It also stops the local variables and the call frame if not displayed correctly, as explained above.

Tip

Switch off this optimization to get faster code. For code density, there are only a few cases that produce slower code. This situation only occurs when other optimizations (such as branch tail merging or peepholing) do not find a pattern.

7.2.3.1.65 -Oi: Inlining

Group

OPTIMIZATIONS

Scope

Compilation unit

Syntax

```
-Oi[=(c<code Size>|OFF)]
```

Arguments

<code Size>: Limit for inlining in code size

OFF: switching off inlining

Default

None. If no <code Size> is specified, the compiler uses a default code size of 40 bytes.

Defines

None

Pragmas

```
#pragma INLINE
```

Description

This option enables inline expansion. If there is a `#pragma INLINE` before a function definition, the Compiler replaces all calls of this function by the code of this function, if possible.

Using the option -Oi=c0 switches off inlining. Functions marked with the #pragma `INLINE` are still inlined. To disable inlining, use the -Oi=OFF option.

Example

```
-Oi


#pragma INLINE


static void f(int i) {


  /* all calls of function f() are inlined */


  /* ... */


}
```

The [`=c<n>`] option extension inlines all functions with a size smaller than `<n>`. For example, compiling with the option `-oi=c100` enables inline expansion for all functions with a size smaller than 100 bytes.

Restrictions

The following functions are not inlined:

- Functions with default arguments
- Functions with labels inside
- Functions with an open parameter list (" `void f(int i,...);`")
- Functions with inline assembly statements
- Functions using local static objects

7.2.3.1.66 -Oilib: Optimize Library Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Oilib[=<arguments>]
```

Arguments

`<arguments>` are one or several of the following suboptions:

`b:` inline calls to `strlen()`

`d:` inline calls to `fabs()` or `fabsf()`

`e:` inline calls to `memset()`

`f:` inline calls to `memcpy()`

`g:` replace shifts left of 1 by array lookup

Default

None

Defines

None

Pragmas

None

Description

This option enables the compiler to optimize specific known library functions to reduce execution time. The Compiler frequently uses small functions such as `strcpy()`, `strcmp()`, and so forth. The following functions are optimized:

- `strcpy()` (only available for ICG-based backends)
- `strlen()` (for example, `strlen("abc")`)
- `abs()` or `fabs()` (for example, `f = fabs(f);')
- `memset()`

  `memset()` is optimized only if:

  - the result is not used
  - `memset()` is used to zero out
  - the size for the zero out is in the range 1 to `0xff`
  - the ANSI library header file `<string.h>` is included

    The following code optimizes `memset ()`

    ```
    (void)memset(&buf, 0, 50);
    ```

    In this case, `_memset_clear_8bitCount`, present in the ANSI library (`string.c`), replaces the call to `memset()`.

- `memcpy()`

  `memcpy()` is optimized only if:

  - the result is not used,
  - the size for the copy out is in the range 0 - 0xff,
  - the ANSI library header file `<string.h>` is included.

    The following code optimizes `memcpy ()`

    ```
    (void)memcpy(&buf, &buf2, 30);
    ```

    In this case, `_memcpy_8bitCount`, present in the ANSI library (`string.c`), replaces the call to `memcpy()`.

`_PowOfTwo_8[val]` replaces `(char)1 << val` if `_PowOfTwo_8` is known at compile time. Similarly, for 16-bit and for 32-bit shifts, the arrays `_PowOfTwo_16` and `_PowOfTwo_32` are used. These constant arrays contain the values 1, 2, 4, 8.... They are declared in `hidef.h`. The compiler performs this optimization only when optimizing for time.

`Using-Oilib` without arguments optimizes calls to all supported library functions.

Example

The example code below compiles the function `f` with the `-Oilib=a` option (only available for ICG-based backends)

```
void f(void) {

  char *s = strcpy(s, ct);

}
```

This translates in a similar fashion to the following function:

```
void g(void) {

  s2 = s;

  while(*s2++ = *ct++);

}
```

See also

-Oi: Inlining

Message C5920

7.2.3.1.67 -Ol: Try to Keep Loop Induction Variables in Registers

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ol<number>
```

Arguments

<number>: number of registers to be used for induction variables

Default

None

Defines

None

Pragmas

None

Description

This option limits the number of loop induction variables the Compiler keeps in registers. Specify any number down to zero (no loop induction variables). The compiler reads and writes loop induction variables within the loop (for example, loop counter), and attempts to keep the variables in registers to reduce execution time and code size. The Compiler takes the optimal number (code density) when this option is not specified. Specifying a high number of loop induction variables may increase code size, particularly for spill and merge code.

NOTE

Disable this option (with -Ol0) if problems occur while debugging. This optimization may increase code complexity when using High-Level Languages, making debugging more difficult.

Listing: With the -Ol0 Option (No Optimization, Pseudo Code) and Listing: Without Option (Optimized, Pseudo Assembler) use the example in Listing: Example (Abstract Code).

Listing: Example (Abstract Code)

```
void main(char *s) {
  do {

    *s = 0;

  } while (*++s);

}
```

The following listing shows pseudo disassembly with the -Ol0 option:

Listing: With the -Ol0 Option (No Optimization, Pseudo Code)

```
loop:
  LD  s, Reg0

  ST  #0, [Reg0]

  INC Reg0

  ST  Reg0, s

  CMP [Reg0],#0

  BNE loop
```

The following listing shows pseudo disassembly without the `-Ol` option (i.e., optimized). Loads and stores from or to variable `s` disappear.

Listing: Without Option (Optimized, Pseudo Assembler)

```
loop:
  ST  #0, s

  INC s

  CMP s,#0

  BNE loop
```

Example

```
  -Ol1
```

7.2.3.1.68 -Ona: Disable Alias Checking

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
  -Ona
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

After optimization, the Compiler redefines variables that may be written by a pointer indirection or an array access. This option prevents the Compiler from redefining these variables, which allows you to reuse already-loaded variables or equivalent constants. Use this option only when you are sure no real writes of aliases to a variable memory location will occur.

Example

Do not compile with `-Ona`.

```
void main(void) {

  int a = 0, *p = &a;

  *p = 1; // real write by the alias *p

  if(a == 0) Error(); // if -Ona is specified,

  // Error() is called!

}
```

7.2.3.1.69 -OnB: Disable Branch Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OnB[=<option Char>{<option Char>}]
```

Arguments

Use one of the following arguments for `<option Char>`:

`a`: Disable short BRA optimization

`b`: Disable Branch JSR to BSR optimization

`d`: Disable dead code optimization

`l`: Disable long branch optimization

`r`: Disable Branch to RTS optimization

`t`: Disable Branch tail optimization

Default

None

Defines

None

Pragmas

None

Description

See Backend for details

Example

```
-OnB
```

Disables all branch optimizations

Seealso

Branch Optimizations

7.2.3.1.70 -Onbf: Disable Optimize Bitfields

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Onbf
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prevents the Compiler from combining sequences of bitfield assignments containing constants. This simplifies debugging and makes the code more readable.

Example
Listing: Example Bitfield Definition

```
struct {
  b0:1;

  b1:1;

  b2:1;

} bf;

void main(void) {
```

```
  bf.b0 = 0;
  bf.b1 = 0;
  bf.b2 = 0;
}
```

Without `-Onbf` (pseudo intermediate code):

```
  BITCLR  bf, #7  // all 3 bits (the mask is 7)


                  // are cleared
```

With `-Onbf` (pseudo intermediate code):

```
  BITCLR  bf, #1  // clear bit 1 (mask 1)


  BITCLR  bf, #2  // clear bit 2 (mask 2)


  BITCLR  bf, #4  // clear bit 3 (mask 4)
```

Example

```
  -Onbf
```

7.2.3.1.71 -Onbt: Disable ICG Level Branch Tail Merging

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
  -Onbt
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option switches the ICG level branch tail merging off. This simplifies debugging and produces more readable code.

The example in the following listing disassembles to the following pseudocode.

Listing: Example Function

```
void main(void) {
  if(x > 0) {

    y = 4;

  } else {

    y = 9;

  }
}
```

Without `-Onbt`, the above example disassembles as shown in the following listing.

Listing: Case (1) Without -Onbt: (Pseudo Intermediate Code)

```
             CMP   x, 0
             BLE   else_label

             LOAD  reg, #4

             BRA   branch_tail

else_label:  LOAD  reg, #9

branch_tail: STORE y, reg

go_on:        ...
```

With the -Obnt compiler option, Listing: Example Function listing disassembles as in the following listing.

Listing: Case (2) with -Onbt: (Pseudo Intermediate Code)

```
            CMP   x, 0
            BLE   else_label

            LOAD  reg, #4

            STORE y, reg

            BRA   go_on

else_label: LOAD  reg, #9

            STORE y, reg

go_on:      ...
```

Example

```
-Onbt
```

7.2.3.1.72 -Onca: Disable any Constant Folding

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Onca
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prevents the Compiler from folding constants over statement boundaries. All arithmetical operations are coded. This option must be set when using the library functions `setjmp()` and `longjmp()`, or the Compiler makes wrong assumptions (refer the following listing).

Listing: Example with If Condition Always True

```
void main(void) {
  jmp_buf env;

  int k = 0;

  if (setjmp(env) == 0) {

    k = 1;

    longjmp(env, 0);

    Err(1);

  } else if (k != 1) {/* assumed always TRUE */

    Err(0);

  }

}
```

Example

```
  -Onca
```

7.2.3.1.73 -Oncn: Disable Constant Folding in Case of a New Constant

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Oncn
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prevents the Compiler from folding constants when the resulting constant is new. The option affects only those processors where constants are difficult to load (for example, RISC processors). On other processors this option makes no change.

Listing: Example (Pseudo Code)

```
void main(void) {
  int a = 1, b = 2, c, d;

  c = a + b;

  d = a * b;

}
```

Case (1) without the `-Oncn` option (pseudo code):

```
  a MOVE  1


  b MOVE  2


  c MOVE  3


  d MOVE  2
```

Case (2) with the -Oncn option (pseudo code):

```
a MOVE  1

b MOVE  2

c ADD   a,b

d MOVE  2
```

The constant 3 is a new constant that does not appear in the source. The constant 2 is already present, so it is still propagated.

Example

```
-Oncn
```

7.2.3.1.74 -OnCopyDown: Do Generate Copy Down Information for Zero Values

Group

OPTIMIZATIONS

Scope

Compilation unit

Syntax

```
-OnCopyDown
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Startup code usually clears all global variables to zero (zero out) and copies any non-zero initialization values to the variables (copy down). Because of this, it is not necessary to copy zero values unless the usual startup code is modified. If a modified startup code contains a copy down but not a zero out, use this option to prevent the compiler from removing the initialization.

NOTE

The case of a copy down without a zero out is unusual. Because the copy down needs much more space than the zero out, code usually contains copy down and zero out, zero out alone, or neither.

In the HIWARE format, the object-file format permits the Compiler to remove single assignments in a structure or array initialization. In the ELF format, the Compiler optimizes only when the whole array or structure is initialized with zero.

NOTE

This option controls the optimizations done in the compiler. However, the linker might further optimize the copy down or the zero out.

Example

```
int i=0;


int arr[10]={1,0,0,0,0,0,0,0,0,0};
```

Using this option, the compiler does not generate a copy down for `i`.

The initialization with zero optimization shown for the `arr` array only works in the HIWARE format. The ELF format requires initializing the whole array to zero.

7.2.3.1.75 -OnCstVar: Disable CONST Variable by Constant Replacement

Group

OPTIMIZATIONS

Scope

Compilation Unit

Syntax

```
-OnCstVar
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option allows you to switch OFF the replacement of CONST variable by the constant value.

Example

```
const int MyConst = 5;


int i;


void myfun(void) {


  i = MyConst;
```

```
}
```

Without the `-OnStVar` option, the compiler replaces each occurrence of `MyConst` with its constant value 5; that is, `i = MyConst` becomes `i = 5`;. The Compiler optimizes the Memory or ROM needed for the `MyConst` constant variable as well. Setting the `-OnCstVar` option avoids this optimization. Use this option only if you want unoptimized code.

7.2.3.1.76 -One: Disable any Low Level Common Subexpression Elimination

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-One
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prevents the Compiler from reusing common subexpressions, such as array indexes and array base addresses. The code size may increase. The low-level CSE does not have the alias problems of the frontend CSE and is therefore switched on by default.

The two CSE optimizations do not cover the same cases. The low-level CSE has a finer granularity but does not handle all cases of the frontend CSE.

Use this option only to generate more readable code for debugging.

Listing: Example (Abstract Code)

```
void main(int i) {
  int a[10];

  a[i] = a[i-1];

}
```

Listing: Case (1) Without Option (Optimized) listing shows a case that does not use `-One` and Listing: Case (2) -One (Not Optimized, Readable) shows a case that uses `-One`.

Listing: Case (1) Without Option (Optimized)

```
tmp1    LD    i
tmp2    LSL   tmp1,#1

tmp3    SUB   tmp2,#2

tmp4    ADR   a

tmp5    ADD   tmp3, tmp4

tmp6    LD    (tmp5)

2(tmp5) ST    tmp6
```

Listing: Case (2) -One (Not Optimized, Readable)

```
tmp1    LD    i
tmp2    LSL   tmp1,#1

tmp3    SUB   tmp2,#2

tmp4    ADR   a

tmp5    ADD   tmp3,tmp4

tmp6    LSL   tmp1,#1   ;calculated twice

tmp7    ADR   a         ;calculated twice

tmp8    ADD   tmp6,tmp7

tmp9    LD    (tmp5)

(tmp8)  ST    tmp9
```

Example

```
-One
```

7.2.3.1.77 -OnP: Disable Peephole Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OnP[=<option Char>{<option Char>}]
```

Arguments

Use one of the following arguments for `<option Char>`:

`a`: Disable peephole combine `AI(S|X)` optimization

`b`: Disable peephole handle constant argument optimization

`c`: Disable peephole `PSH/PUL` instead `AIS` optimization

`d`: Disable peephole combine bit operations optimization

`e`: Disable peephole combine bit set/clr optimization

`f`: Disable peephole `PSH PUL` optimization

`g`: Disable peephole `RTS RTS` optimization

`h`: Disable peephole unused loads optimization

`i`: Disable peephole unused stores optimization

`j`: Disable peephole unused compares optimization

`k`: Disable peephole unnecessary tests optimization

`l`: Disable peephole unnecessary transfers optimization

`m`: Disable peephole `JSR` to `JMP` optimization

`n`: Disable peephole `CMP #1` optimization

`o`: Disable peephole simple inline assembler optimizations

`x`: Disable peephole load immediate to HX [HCS08 only]

Default

None

Defines

None

Pragmas

None

Description

If `-OnP` is specified, the Compiler disables the whole peephole optimizer. To disable only a single peephole optimization, use the optional syntax `-OnP=<char>`. For example:

Suboption `-OnP=m` (Disable peephole `JSR` to JMP optimization)

The compiler replaces a `JSR`- `RTS` sequence with a single `JMP` instruction. This saves single-byte code and 2-byte stack spaces. To avoid this optimization, use the `-onp=m` option.

With `-onp=m`:

```
...


JSR   Subroutine


RTS
```

Without `-onp=m`:

```
...


JMP   Subroutine
```

Example

```
-OnP=m
```

Disables the peephole optimization `m`

7.2.3.1.78 -OnPMNC: Disable Code Generation for NULL Pointer to Member Check

Group

OPTIMIZATIONS

Scope

Compilation Unit

Syntax

```
-OnPMNC
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Before assigning a pointer to a member in C++, you must ensure that the pointer to the member is not NULL in order to generate correct and safe code. In embedded systems development, the difficulty becomes generating the denser code while avoiding overhead whenever possible (this NULL check code is a good example). This option enables you to switch off the code generation for the NULL check.

Example

```
-OnPMNC
```

7.2.3.1.79 -Ont: Disable Tree Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ont[={%|&|*|+|-|/|0|1|7|8|9|?|^|a|b|c|d|e| f|h|i|l|m|n|o|p|q|r|s|t|u|v|w|||~}]
```

Arguments

`%`: Disable mod optimization

`&`: Disable bit and optimization

`*`: Disable mul optimization

`+`: Disable plus optimization

`-`: Disable minus optimization

`/`: Disable div optimization

`0`: Disable and optimization

`1`: Disable or optimization

`7`: Disable extend optimization

`8`: Disable switch optimization

`9`: Disable assign optimization

`?`: Disable test optimization

`a`: Disable statement optimization

`b`: Disable constant folding optimization

`c`: Disable compare optimization

`d`: Disable binary operation optimization

`e`: Disable constant swap optimization

`f`: Disable condition optimization

`g`: Disable compare size optimization

`h`: Disable unary minus optimization

`i`: Disable address optimization

`j`: Disable transformations for inlining

`l`: Disable label optimization

`m`: Disable left shift optimization

`n`: Disable right shift optimization

`o`: Disable cast optimization

`p`: Disable cut optimization

`q`: Disable 16-32 compare optimization

`r`: Disable 16-32 relative optimization

`s`: Disable indirect optimization

`t`: Disable for optimization

`u`: Disable while optimization

`v`: Disable do optimization

`w`: Disable if optimization

`^`: Disable exor optimization

`|`: Disable bit or optimization

`~`: Disable bit neg optimization

Default

Specifying `-Ont` with no arguments disables all optimizations

Defines

None

Pragmas

None

Description

The Compiler contains a special optimizer which optimizes the internal tree data structure. This tree data structure holds the semantic of the program and represents the parsed statements and expressions.

This option disables the tree optimizer. Use this option for debugging and to force the Compiler to produce `straightforward' code. Note that the optimizations below are just examples for the classes of optimizations.

Using this option with the arguments below disables the optimizations.

`-Ont=~` disables optimization of `~~i' into `i'

`-Ont=|` disables optimization of ` i|0xffff' into ` 0xffff'

`-Ont=w` disables optimization of ` if (1) i = 0;' into `i = 0;'

`-Ont=v` disables optimization of ` do ... while(0)' into `...'

`-Ont=u` disables optimization of ‘while (cond) break;' into ‘cond;', provided there are no labels within the 'while' statement list.

`-Ont=t` disables optimization of ` for(;;) ...' into ` while(1) ...'

`-Ont=s` disables optimization of ` *&i' into ` i'

`-Ont=r` disables optimization of ` L<=4' into 16-bit compares

`-Ont=q` disables reduction of long compares into int compares. For example:

`if (uL == 0)` optimizes into:

```
if ((int)(uL>>16) == 0 && (int)uL == 0)
```

`-Ont=p` disables optimization of ` (char)(long)i' into ` (char)i'

`-Ont=o` disables optimization of ` (short)(int)L' into ` (short)L' if `short` and `int` have the same size.

`-Ont=n` and `-Ont=m` disable optimization of shift optimizations (<<, >>, -Ont=n or -Ont= `m`):

Reduction of shift counts to unsigned `char`:

`uL = uL1 >> uL2;` optimizes into:

```
uL = uL1 >> (unsigned char)uL2;
```

Optimization of zero shift counts:

`uL = uL1 >> 0;` optimizes into:

```
uL = uL1;
```

Optimization of shift counts greater than the object to be shifted:

`uL = uL1 >> 40;` optimizes into:

```
uL = 0L;
```

Strength reduction for operations followed by a cut operation:

`ch = uL1 * uL2;` optimizes into:

```
ch = (char)uL1 * (char)uL2;
```

Replacing shift operations by load or store:

`i = uL >> 16;` optimizes into:

```
i = *(int *)(&uL);
```

Shift count reductions:

`ch = uL >> 17;` optimizes into:

```
ch = (*(char *)(&uL)+1)>>1;
```

Optimization of shift combined with binary and:

`ch = (uL >> 25) & 0x10;` optimizes into:

```
ch = ((*(char *)(&uL))>>1) & 0x10;
```

`-Ont=l` disables optimization removal of labels if not used

`-Ont=i` disables optimization of `&*p' into `p'

`-Ont=j` transforms the syntax tree into an equivalent form which allows more inlining cases. Enable inlining to use this option.

`-Ont=h` disables optimization of `-(-i)' into `i'

`-Ont=f` disables optimization of `(a==0)' into `(!a)'

`-Ont=e` disables optimization of `2*i' into `i*2'

`-Ont=d` disables optimization of `us & ui' into `us& (unsigned short)ui'

`-Ont=c` disables optimization of `if ((long)i)' into `if (i)'

`-Ont=b` disables optimization of `3+7' into `10'

`-Ont=a` disables optimization of last statement in function if result is not used

`-Ont=^` disables optimization of `i^0' into `i'

`-Ont=?` disables optimization of `i = (int)(cond ? L1:L2);' into `i = cond ? (int)L1:(int)L2;'

`-Ont=9` disables optimization of `i=i;'

`-Ont=8` disables optimization of empty switch statement

`-Ont=7` disables optimization of `(long)(char)L' into `L'

`-Ont=1` disables optimization of `` `a || 0' `` into `` `a' ``

`-Ont=0` disables optimization of `` `a && 1' `` into `` `a' ``

`-Ont=/` disables optimization of `` `a/1' `` into `` `a' ``

`-Ont=-` disables optimization of `` `a-0' `` into `` `a' ``

`-Ont=+` disables optimization of `` `a+0' `` into `` `a' ``

`-Ont=*` disables optimization of `` `a*1' `` into `` `a' ``

`-Ont=&` disables optimization of `` `a&0' `` into `` `0' ``

`-Ont=%` disables optimization of `` `a%1' `` into `` `0' ``

Example

```
fibo.c -Ont
```

7.2.3.1.80 -OnX: Disable Frame Pointer Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OnX
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Using this option prevents the Compiler from converting stack pointer-relative accesses into X-relative accesses. The frame optimizer tries to convert all SP-relative accesses (local variables, spills) into shorter and faster X-relative accesses. In addition, the Compiler traces the value of `H:X` and removes useless `TSX` and `AIX` instructions. Using `-OnX` to switch the frame optimizer off facilitates debugging.

Example

```
-OnX
```

Disables frame pointer optimizations

7.2.3.1.81 -Or: Allocate Local Variables into Registers

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Or
```

Arguments

None

Default

None

Defines

```
__OPTIMIZE_REG__
```

Pragmas

None

Description

Allocate local variables (`char` or `int`) in registers. The number of local variables allocated in registers depends on the number of available registers. Use this option when using variables as loop counters or switch selectors or when the processor requires register operands for multiple operations (for example, RISC processors). Compiling with this option may increase your code size (spill and merge code).

NOTE

This optimization may increase code complexity when using High-Level Languages, making debugging more difficult.

NOTE

For some backends this option has no effect and code does not change.

Listing: Allocate Local Variables into Registers Example

```
-Or
int main(void) {

  int a,b;

  return a + b;

}

Case (1) without option -Or (pseudo code):

tmp1 LD    a

tmp2 LD    b

tmp3 ADD   tmp1,tmp2

     RET   tmp3

Case (2)with option -Or (pseudo code):

tmp1 ADD a,b

RET tmp1
```

7.2.3.1.82 -Ous, -Ou, and -Onu: Optimize Dead Assignments

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-O(us|u|nu)
```

Arguments

`us`: yes, but never if HLI present in function (default)

`u`: always (even if HLI is present in function)

`nu`: never

Default

`-Ous`: Optimization enabled for functions containing no inline assembler code

Defines

None

Pragmas

None

Description

Optimizes dead assignments. The Compiler removes assignments to unused local variables.

There are three possible settings for this option: `-Ou` is given, `-Onu` is given, or `-Ous` is given:

`-Ou`: Always optimize dead assignments (even if HLI is present in current function). The Compiler does not consider inline assembler accesses.

NOTE

This option is unsafe when inline assembler code contains accesses to local variables.

`-Onu`: No optimization occurs. This generates the best possible debug information, and produces larger and slower code.

`-Ous`: Optimize dead assignments if HLI is not present in the current function.

NOTE

The compiler is not aware of `longjmp()` or `setjmp()` calls. These and similar functions may generate a control flow which the Compiler does not recognize. Therefore, either do not use local

variables in functions using `longjmp()` or `setjmp()` or use `-Onu` to switch this optimization off.

NOTE

Dead assignments to `volatile` declared global objects are never optimized.

Example

```
void main(int x) {

  f(x);

  x = 1; /* this assignment is dead and is

            removed if -Ou is active */

}
```

7.2.3.1.83 -Pe: Do Not Preprocess Escape Sequences in Strings with Absolute DOS Paths

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-Pe
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The Compiler handles escape sequences in macros in an include directive, similar to the way the Compiler handles escape sequences in a `printf()` instruction:

```
#define STRING "C:\myfile.h"


#include STRING
```

produces an error:

```
>> Illegal escape sequence
```

and:

```
printf(STRING);
```

produces a carriage return with line feed:

```
C:


myfile
```

Using the `-Pe` option, the Compiler ignores escape sequences in strings that contain a DOS drive letter (' `a`' - ' `z`', ' `A`' - ' `Z`') followed by a colon ' `:`' and a backslash ' `\`'.

Enable the `-Pe` option to make the Compiler handle strings in include directives differently from other strings. Escape sequences in include directive strings are not evaluated.

The following example:

```
#include "C:\names.h"
```

results in exactly the same include filename as in the source file (" `C:\names.h`"). If the filename appears in a macro, the Compiler does not distinguish between filename usage and normal string usage with escape sequence. This occurs because the macro `STRING` has to be the same for the include and the printf() call, as shown below:

```
#define STRING "C:\n.h"


#include STRING /* means: "C:\n.h" *


void main(void) {


  printf(STRING);/* means: "C:", new line and ".h" */


}
```

Use this option to use macros for include files. This prevents escape sequence scanning in strings that start with a DOS drive letter ('a - 'z', 'A' - 'Z') followed by a colon ' `:`' and a backslash ' `\`'. With the option set, the above example includes the ' `C:\n.h`' file and calls `printf()` with `"C:\n.h"`).

Example

```
-Pe
```

7.2.3.1.84 -Pio: Include Files Only Once

Group

INPUT

Scope

Compilation Unit

Syntax

```
-Pio
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Includes every header file only once. Whenever the compiler reaches an `#include` directive, the compiler checks the named file against the files already included. If the file has already been read, the compiler ignores the `#include` directive. It is common practice to protect header files from multiple inclusion by conditional compilation, as shown below:

```
/* Header file myfile.h */

#ifndef _MY_FILE_H_

#define _MY_FILE_H_

/* .... content .... */

#endif /* _MY_FILE_H_ */
```

When `#ifndef` and `#define` directives are issued, the Compiler reads header file content only once even when the header file is included several times. This solves many problems as C-language protocol does not allow you to define structures (such as enums or typedefs) more than once.

Using this option to protect all header files can safely accelerate compilation.

Do not use this option when a header file *must* be included twice, for example, the file contains macros which are set differently at different inclusion times. In those instances, use #pragma ONCE: Include Once to accelerate the inclusion of safe header files which do not contain macros of that nature.

Example

```
-Pio
```

See also

#pragma ONCE: Include Once

7.2.3.1.85 -Prod: Specify Project File at Startup

Group

STARTUP

This option cannot be specified interactively.

Scope

None

Syntax

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

Arguments

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

Default

None

Defines

None

Pragmas

None

Description

This option can only be specified at the command line while starting the application. It cannot be specified in any other circumstances.

Use this option to make the application open the file as a configuration file. When `<file>` names a directory instead of a file, the application appends the default name `project.ini`. When loading fails, a message box appears.

Example

```
compiler.exe -prod=project.ini
```

(Use the compiler executable name instead of `"compiler"`.)

See also

Local Configuration File

7.2.3.1.86 -Qvtp: Qualifier for Virtual Table Pointers

Group

CODE GENERATION

Scope

Application

Syntax

```
-Qvtp(none|far|near)
```

Arguments

None

Default

```
-Qvptnone
```

Defines

None

Pragmas

None

Description

Using a virtual function in C++ requires an additional pointer to virtual function tables. The Compiler cannot access the pointer and generates the pointer in every class object when virtual function tables are associated.

NOTE

Specifying qualifiers not supported by the Backend or CPU has no effect (refer Backend).

Example

```
-QvtpFar
```

This sets the qualifier for virtual table pointers to `__far`, which enables placement of the virtual tables into a `__FAR_SEG` segment (if the Backend or CPU supports `__FAR_SEG` segments).

7.2.3.1.87 -Rp (-Rpe, -Rpt): Large Return Value Type

Group

OPTIMIZATIONS

Scope

Application

Syntax

```
-Rp(t|e)
```

Arguments

`t`: Pass large return value by pointer, always with temporary

`e`: Pass large return value by pointer and temporary elimination

Default

```
-Rpe
```

Defines

None

Pragmas

None

Description

The Compiler supports this option even though returning a `large' return value may not be as efficient as using an additional pointer. The Compiler introduces an additional parameter for the return value if the return value cannot be passed in registers.

Consider the code in the following listing.

Listing: Example Code

```
typedef struct { int i[10]; } S;
S F(void);

S s;

void main(void) {

  s = F();

}
```

In the above case, with `-Rpt`, the code will appear as in the following listing.

Listing: Pass Large Return Value by Pointer

```
void main(void) {
  S tmp;

  F(&tmp);

  s = tmp; /* struct copy */

}
```

The above approach is always correct but not efficient. Instead, pass the destination address directly to the callee. This makes it unnecessary to declare a temporary and a struct copy in the caller (i.e., `-Rpe`), as shown in the following listing.

Listing: Pass Large Return Value by Pointer and Temporary Elimination

```
void main(void) {
```

```
  F(&s);
}
```

The above example may produce incorrect results for rare cases (for example, if the `F()` function returns something overlapping `s`). Because the Compiler does not detect such rare cases, two options are provided: the `-Rpt` (always correct, but inefficient), or `-Rpe` (efficient) options.

7.2.3.1.88 -T: Flexible Type Management

Group

LANGUAGE.

Scope

Application

Syntax

```
  -T<Type Format>
```

Arguments

`<Type Format>`: See below

Default

Depends on target, see Compiler Backend

Defines

To deal with different type sizes, the Compiler predefines one of the following define groups (refer the following listing).

Listing: Define Groups

```
__CHAR_IS_SIGNED__
__CHAR_IS_UNSIGNED__

__CHAR_IS_8BIT__

__CHAR_IS_16BIT__

__CHAR_IS_32BIT__

__CHAR_IS_64BIT__
```

__SHORT_IS_8BIT__

__SHORT_IS_16BIT__

__SHORT_IS_32BIT__

__SHORT_IS_64BIT__

__INT_IS_8BIT__

__INT_IS_16BIT__

__INT_IS_32BIT__

__INT_IS_64BIT__

__ENUM_IS_8BIT__

__ENUM_IS_16BIT__

__ENUM_IS_32BIT__

__ENUM_IS_64BIT__

__ENUM_IS_SIGNED__

__ENUM_IS_UNSIGNED__

__PLAIN_BITFIELD_IS_SIGNED__

__PLAIN_BITFIELD_IS_UNSIGNED__

__LONG_IS_8BIT__

__LONG_IS_16BIT__

__LONG_IS_32BIT__

__LONG_IS_64BIT__

__LONG_LONG_IS_8BIT__

__LONG_LONG_IS_16BIT__

__LONG_LONG_IS_32BIT__

__LONG_LONG_IS_64BIT__

__FLOAT_IS_IEEE32__

__FLOAT_IS_IEEE64__

__FLOAT_IS_DSP__

__DOUBLE_IS_IEEE32__

__DOUBLE_IS_IEEE64__

__DOUBLE_IS_DSP__

__LONG_DOUBLE_IS_IEEE32__

__LONG_DOUBLE_IS_IEEE64__

__LONG_DOUBLE_IS_DSP__

__LONG_LONG_DOUBLE_IS_IEEE32__

__LONG_LONG_DOUBLE_IS_IEEE64__

__LONG_LONG_DOUBLE_DSP__

__VTAB_DELTA_IS_8BIT__

__VTAB_DELTA_IS_16BIT__

__VTAB_DELTA_IS_32BIT__

__VTAB_DELTA_IS_64BIT__

__PTRMBR_OFFSET_IS_8BIT__

__PTRMBR_OFFSET_IS_16BIT__

__PTRMBR_OFFSET_IS_32BIT__

__PTRMBR_OFFSET_IS_64BIT__

Pragmas

None

Description

This option allows configurable type settings. The syntax of the option is: `-T{<type><format>}`

NOTE

Flexible type management does not support 8-bit `int` definitions. When generating code, use minimal startup code instead of ANSI startup code.

For `<type>`, specify one of the keys listed in the following table.

Table 7-10. Data Type Keys

Type	Key
`char`	`'c'`
`short`	`'s'`
`int`	`'i'`
`long`	`'L'`
`long long`	`'LL'`
`float`	`'f'`
`double`	`'d'`
`long double`	`'Ld'`
`long long double`	`'LLd'`
`enum`	`'e'`
`sign plain bitfield`	`'b'`
`virtual table delta size`	`'vtd'`
`pointer` to member offset size	`'pmo'`

NOTE

Keys are not case-sensitive, for example, both `'f'` and `'F'` may be used for the type "`float`".

Change the sign of type `char` or of the enumeration type with a prefix placed before the key for the char key. Refer the following table.

Table 7-11. Keys for Signed and Unsigned Prefixes

Sign Prefix	Key
`signed`	`'s'`
`unsigned`	`'u'`

Change the sign of the plain bitfield type with the options shown in the following table. Plain bitfields are bitfields defined or declared without an explicit signed or unsigned qualifier, for example, `int field:3`. Using this option, you can specify whether to handle the `int` in the previous example as `signed int` or as `unsigned int`.

NOTE

This option may not be available on all targets. Also the default setting may vary. For more information, refer to the topic Sign of Plain Bitfields.

Table 7-12. Keys for Signed and Unsigned Bitfield Prefixes

Sign Prefix	Key
plain signed bitfield	'bs'
plain unsigned bitfield	'bu'

For `<format>`, specify one of the keys in the following table.

Table 7-13. Data Format Specifier Keys

Format	Key
8-bit integral	'1'
16-bit integral	'2'
24-bit integral	'3'
32-bit integral	'4'
64-bit integral	'8'
IEEE32 floating	'2'
IEEE64 floating	'4'
DSP (32-bit)	'0'

Not all formats may be available for a target. See Backend for supported formats.

NOTE

At least one type for each basic size (1, 2, 4 bytes) must be available. You must set a size for all types, at least equal to one. See Backend for default settings.

NOTE

Enumeration types have the type `` `signed int' `` by default for ANSI-C compliance.

Use the `-Tpmo` option to change the pointer to a member offset value type. The default setting is 16 bits. The Compiler uses the pointer to the member offset as a C++ pointer to members only.

`-Tsc` sets ´`char`´ to ´`signed char`´

and `-Tuc` sets ´`char`´ to ´`unsigned char`´

Listing: Example

```
-Tsc1s2i2L4LL4f2d4Ld4LLd4e2 denotes:
  signed char with 8 bits (sc1)

  short and int with 16 bits (s2i2)

  long,long long with 32 bits (L4LL4)

  float with IEEE32 (f2)

  double, long double and long long double with IEEE64 (d4Ld4LL4)

  enum with 16 bits (signed) (e2)
```

Listing: Restrictions

```
For integrity and ANSI compliance, the following rules must be true:
  sizeof(char)        <= sizeof(short)

  sizeof(short)       <= sizeof(int)

  sizeof(int)         <= sizeof(long)

  sizeof(long)        <= sizeof(long long)

  sizeof(float)       <= sizeof(double)

  sizeof(double)      <= sizeof(long double)

  sizeof(long double) <= sizeof(long long double)
```

NOTE

Do not set `char` to 16 bits and `int` to 8 bits.

Change type sizes with caution. Type sizes must be consistent over the whole application. The libraries delivered with the Compiler are compiled with the standard type settings.

Use caution when changing type sizes for under or overflows (for example, assigning a large value to a small object), as shown in the following example:

```
int i; /* -Ti1 int has been set to 8 bits! */



i = 0x1234; /* i will set to 0x34! */
```

Example

Setting the size of `char` to 16 bits:

```
-Tc2
```

Setting the size of `char` to 16 bits and plain char is signed:

```
-Tsc2
```

Setting `char` to 8 bits and unsigned, `int` to 32 bits and `long long` to 32 bits:

```
-Tuc1i4LL4
```

Setting float to IEEE32 and double to IEEE64:

```
-Tf2d4
```

The `-Tvtd` option allows you to change the delta value type inside virtual function tables. The default setting is 16 bits.

You can also set this option using the dialog box in the Graphical User Interface.

See also

Sign of Plain Bitfields

7.2.3.1.89 -V: Prints the Compiler Version

Group

VARIOUS

Scope

None

Syntax

```
-V
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Prints the internal subversion numbers of the Compiler component parts and the location of current directory.

NOTE

This option can determine the current directory of the Compiler.

Example

`-v` produces the following list:

```
Directory: \software\sources\c
ANSI-C Front End, V5.0.1, Date Jan 01 2005
Tree CSE Optimizer, V5.0.1, Date Jan 01 2005
Back End V5.0.1, Date Jan 01 2005
```

7.2.3.1.90 -View: Application Standard Occurrence

Group

HOST

Scope

Compilation Unit

Syntax

```
-View<kind>
```

Arguments

`<kind>` is one of:

Window: Application window has default window size

Min: Application window is minimized

Max: Application window is maximized

Hidden: Application window is not visible (only if arguments)

Default

Application started with arguments: Minimized

Application started without arguments: Window

Defines

None

Pragmas

None

Description

If used with no arguments, this option starts the application as a normal window. If used with arguments (for example, from the maker to compile or link a file), then the application runs minimized to allow batch processing.

Use this option to specify application behavior. Use `-ViewWindow` to view the application with its normal window. Use `-ViewMin` to view the application as an icon in the task bar. Use - `ViewMax` to maximize the application (filling the whole screen). Use `-ViewHidden` to process the application arguments (for example, files to be compiled or linked) completely in the background (no window and no icon visible in the task bar). However, if you use the -N option, a dialog box is still possible (refer -N: Show Notification Box in Case of Errors).

Example

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

7.2.3.1.91 -WErrFile: Create "err.log" Error File

Group

MESSAGES

Scope

Compilation Unit

Syntax

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

Arguments

None

Default

```
-WErrFileOn
```

Defines

None

Pragmas

None

Description

Using this option, the Compiler uses a return code to report errors back to the tools. When errors occur, 16-bit window environments use `err.log` files, containing a list of error numbers, to report the errors. If no errors occur, the 16-bit window environments delete the `err.log` file. Using UNIX or WIN32, the return code makes the `err.log` file unnecessary when only UNIX or WIN32 applications are involved.

NOTE

If you use a 16-bit maker with this tool, you must create the error file to signal any errors.

Example

```
-WErrFileOn
```

This creates or deletes the `err.log` file when the application finishes.

```
-WErrFileOff
```

The application does not modify the existing `err.log` file.

See also

-WStdout: Write to Standard Output

-WOutFile: Create Error Listing File

7.2.3.1.92 -Wmsg8x3: Cut Filenames in Microsoft Format to 8.3

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-Wmsg8x3
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Some editors (early versions of WinEdit) expect the filename in Microsoft message format (8.3 format). That means the filename can have up to eight characters and no more than a three-character extension. Longer filenames are possible when you use Win95 or WinNT. This option truncates the filename to the 8.3 format.

Example

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

Setting the `-Wmsg8x3` option changes the above message to:

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

See also

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

7.2.3.1.93 -WmsgCE: RGB Color for Error Messages

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgCE<
RGB>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option changes the error message color. The specified value must be an RGB (Red/Green/Blue) value and must be specified in decimal.

Example

-WmsgCE255 changes the error messages to blue

7.2.3.1.94 -WmsgCF: RGB Color for Fatal Messages

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgCF<
RGB>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option changes the color of a fatal message. The specified value must be an RGB (Red/Green/Blue) value and must be specified in decimal.

Example

`-WmsgCF255` changes the fatal messages to blue

7.2.3.1.95 -WmsgCI: RGB Color for Information Messages

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgCI<
RGB>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option changes the color of an information message. The specified value must be an RGB (Red/Green/Blue) value and must be specified in decimal.

Example

-WmsgCI255 changes the information messages to blue

7.2.3.1.96 -WmsgCU: RGB Color for User Messages

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgCU<
RGB>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option changes the color of a user message. The specified value must be an RGB (Red/Green/Blue) value and must be specified in decimal.

Example

-WmsgCU255 changes the user messages to blue

7.2.3.1.97 -WmsgCW: RGB Color for Warning Messages

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgCW<
RGB>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option changes the color of a warning message. The specified value must be an `RGB` (Red/Green/Blue) value and must be specified in decimal.

Example

`-WmsgCW0` changes the warning messages to black

7.2.3.1.98 -WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

Group

MESSAGES

Scope

Compilation Unit

Syntax

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

Arguments

`v`: Verbose format

`m`: Microsoft format

Default

```
-WmsgFbm
```

Defines

None

Pragmas

None

Description

Use this option to start the Compiler with additional arguments (for example, files and Compiler options). If you start the Compiler with arguments (for example, from the Make Tool or with the `%f' argument from the CodeWright IDE), the Compiler compiles the files in a batch mode. No Compiler window is visible and the Compiler terminates after job completion.

When compiling in batch mode, the Compiler writes messages to a file instead of to the screen. This file contains only the compiler messages (refer examples below).

In batch mode, the Compiler writes messages (errors, warnings, information messages) using a Microsoft message format.

The v argument changes the default format from the Microsoft format (only line information) to a more verbose error format with line, column, and source information.

NOTE

Using the verbose message format may slow down the compilation.

Example

```
void myfun(void) {


  int i, j;


  for(i=0;i<1;i++);


}
```

The Compiler may produce the following file if it is running in batch mode (for example, started from the Make tool):

```
X:\C.C(3): INFORMATION C2901: Unrolling loop


X:\C.C(2): INFORMATION C5702: j: declared in function
myfun but not referenced
```

Setting the format to verbose stores more information in the file:

```
-WmsgFbv

>> in "X:\C.C", line 3, col 2, pos 33


  int i, j;


  for(i=0;i<1;i++);


  ^


INFORMATION C2901: Unrolling loop

>> in "X:\C.C", line 2, col 10, pos 28


void myfun(void) {


  int i, j;


          ^


INFORMATION C5702: j: declared in function myfun but not
referenced
```

See also

ERRORFILE: Error Filename Specification environment variable

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

7.2.3.1.99 -WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

Group

MESSAGES

Scope

Compilation Unit

Syntax

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

Arguments

v: Verbose format

m: Microsoft format

Default

```
-WmsgFiv
```

Defines

None

Pragmas

None

Description

This option sets the Compiler message format to interactive mode (that is, with a visible window) if the Compiler starts without additional arguments (for example, files and Compiler options).

The Compiler uses the verbose error file format to write the Compiler messages (errors, warnings, information messages).

The m argument changes the format from the default verbose format (with source, line and column information) to the Microsoft format (only line information).

NOTE

Using the Microsoft format may increase compilation speed.

Example

```
void myfun(void) {

  int i, j;

  for(i=0;i<1;i++);

}
```

The Compiler may produce the following error output in the Compiler window if it is running in interactive mode:

```
Top: X:\C.C

Object File: X:\C.O

>> in "X:\C.C", line 3, col 2, pos 33

  int i, j;

  for(i=0;i<1;i++);

  ^

INFORMATION C2901: Unrolling loop
```

Setting the format to Microsoft, less information is displayed:

```
-WmsgFim
```

```
Top: X:\C.C


Object File: X:\C.O


X:\C.C(3): INFORMATION C2901: Unrolling loop
```

See also

ERRORFILE: Error Filename Specification

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

7.2.3.1.100 -WmsgFob: Message Format for Batch Mode

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgFob<string>
```

Arguments

<string>: format string (see below).

Default

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

Defines

None

Pragmas

None

Description

This option modifies the default message format in batch mode. The following table lists supported formats (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

Table 7-14. Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	X:\Freescale\
%f	Path and name	X:\Freescale\mysourcefile
%n	filename	mysourcefile
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

Example

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

Produces a message in the following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

ERRORFILE: Error Filename Specification

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for No Position Information

7.2.3.1.101 -WmsgFoi: Message Format for Interactive Mode

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgFoi<string>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

This option modifies the default message format in interactive mode. The following table lists supported formats (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

Table 7-15. Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	X:\sources\
%f	Path and name	X:\sources\mysourcefile
%n	filename	mysourcefile
%e	Extension	.cpph
%N	File (8 chars)	mysource

Table continues on the next page...

Table 7-15. Message Format Specifiers (continued)

Format	Description	Example
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space.	
%'	A ' if the filename, the path or the extension contains a space	

Example

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

Produces a message in the following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

ERRORFILE: Error Filename Specification

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for No Position Information

-WmsgFob: Message Format for Batch Mode

7.2.3.1.102 -WmsgFonf: Message Format for No File Information

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgFonf<string>
```

Arguments

<string>: format string (See below.)

Default

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

Defines

None

Pragmas

None

Description

Sometimes there is no file information available for a message (for example, when a message is unrelated to a specific file). Then <string> defines the message format string to use. The following table lists the supported formats.

Table 7-16. Message Format Specifiers

Format	Description	Example
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

Example

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

Produces a message in the following format:

```
information L10324: Linking successful
```

See also

ERRORFILE: Error Filename Specification

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for No Position Information

-WmsgFoi: Message Format for Interactive Mode

7.2.3.1.103 -WmsgFonp: Message Format for No Position Information

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgFonp<string>
```

Arguments

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

Default

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

Defines

None

Pragmas

None

Description

Sometimes there is no position information available for a message (for example, when a message is unrelated to a certain position). Then `<string> defines` the message format string to use. The following table lists the supported formats.

Table 7-17. Message Format Specifiers

Format	Description	Example
%K	Uppercase type	ERROR
%k	Lowercase type	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path, or the extension contains a space	

Example

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

Produces a message in the following format:

```
information L10324: Linking successful
```

Seealso

ERRORFILE: Error Filename Specification

-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFonp: Message Format for No Position Information

-WmsgFoi: Message Format for Interactive Mode

7.2.3.1.104 -WmsgNe: Maximum Number of Error Messages (enter <number>)

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-WmsgNe<number>
```

Arguments

`<number>`: Maximum number of error messages

Default

`50`

Defines

None

Pragmas

None

Description

This option sets the maximum number of error messages that display while the Compiler is processing.

NOTE

Subsequent error messages which depend upon a previous error message may not process correctly.

Example

```
-WmsgNe2
```

Stops compilation after two error messages

See also

-WmsgNi: Maximum Number of Information Messages (enter <number>)

-WmsgNw: Maximum Number of Warning Messages (enter <number>)

7.2.3.1.105 -WmsgNi: Maximum Number of Information Messages (enter <number>)

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-WmsgNi<number>
```

Arguments

`<number>`: Maximum number of information messages

Default

`50`

Defines

None

Pragmas

None

Description

This option limits the number of information messages that the Compiler logs.

Example

```
-WmsgNi10
```

Ten information messages logged

See also

-WmsgNe: Maximum Number of Error Messages (enter <number>)

-WmsgNw: Maximum Number of Warning Messages (enter <number>)

7.2.3.1.106 -WmsgNu: Disable User Messages

Group

MESSAGES

Scope

None

Syntax

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

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

t: Disable type of messages

Default

None

Defines

None

Pragmas

None

Description

The application produces messages that are not in the following normal message categories: WARNING, INFORMATION, ERROR, or FATAL. This option disables user messages and allows only those messages in the normal message categories (WARNING, INFORMATION, ERROR, or FATAL). This reduces the number of messages, and simplifies the error parsing of other tools.

a: Disables the application from generating information about all included files.

b: Disables messages about reading files (for example, the files used as input).

c: Disables messages informing about generated files.

d: Disables information about statistics (for example, code size, RAM or ROM usage and so on).

e: Disables informal messages (for example, memory model, floating point format, etc.).

NOTE

Depending on the application, the Compiler may not recognize all suboptions. For compatibility, the Compiler ignores unrecognized suboptions.

Example

```
-WmsgNu=c
```

7.2.3.1.107 -WmsgNw: Maximum Number of Warning Messages (enter <number>)

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-WmsgNw<number>
```

Arguments

`<number>`: Maximum number of warning messages

Default

`50`

Defines

None

Pragmas

None

Description

This option sets the number of warning messages.

Example

```
-WmsgNw15
```

Fifteen warning messages logged

See also

-WmsgNe: Maximum Number of Error Messages (enter <number>)

-WmsgNi: Maximum Number of Information Messages (enter <number>) i

7.2.3.1.108 -WmsgSd: Setting a Message to Disable

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSd<number>
```

Arguments

`<number>`: Message number to be disabled, for example, 1801

Default

None

Defines

None

Pragmas

None

Description

This option prevents messages from appearing in the error output. This option cannot be used in #pragma OPTION: Additional Options. Use this option only with #pragma MESSAGE: Message Setting.

Example

```
-WmsgSd1801
```

Disables message for implicit parameter declaration

See also

-WmsgSe: Setting a Message to Error

-WmsgSi: Setting a Message to Information

-WmsgSw: Setting a Message to Warning

7.2.3.1.109 -WmsgSe: Setting a Message to Error

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSe<number>
```

Arguments

`<number>`: Message number to be an error, for example, `1853`

Default

None

Defines

None

Pragmas

None

Description

This option changes a message to an error message. This option cannot be used in #pragma OPTION: Additional Options. Use this option only with #pragma MESSAGE: Message Setting.

Example

```
COMPOTIONS=-WmsgSe1853
```

See also

-WmsgSd: Setting a Message to Disable

-WmsgSi: Setting a Message to Information

-WmsgSw: Setting a Message to Warning

7.2.3.1.110 -WmsgSi: Setting a Message to Information

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSi<number>
```

Arguments

`<number>`: Message number to be an information, for example, `1853`

Default

None

Defines

None

Pragmas

None

Description

This option sets a message to an information message. This option cannot be used with #pragma OPTION: Additional Options. Use this option only with #pragma MESSAGE: Message Setting.

Example

```
-WmsgSi1853
```

See also

-WmsgSd: Setting a Message to Disable

-WmsgSe: Setting a Message to Error

-WmsgSw: Setting a Message to Warning

7.2.3.1.111 -WmsgSw: Setting a Message to Warning

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSw<number>
```

Arguments

`<number>`: Error number to be a warning, for example, `2901`

Default

None

Defines

None

Pragmas

None

Description

This option sets a message to a warning message. This option cannot be used with #pragma OPTION: Additional Options. Use this option only with #pragma MESSAGE: Message Setting.

Example

```
-WmsgSw2901
```

See also

-WmsgSd: Setting a Message to Disable

-WmsgSe: Setting a Message to Error

-WmsgSi: Setting a Message to Information i

7.2.3.1.112 -WOutFile: Create Error Listing File

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-WOutFile(On|Off)
```

Arguments

None

Default

Error listing file is created

Defines

None

Pragmas

None

Description

This option controls whether the Compiler creates an error listing file. The error listing file contains a list of all messages and errors that occur during processing. The name of the listing file is controlled by the ERRORFILE: Error Filename Specification environment variable.

NOTE

You can also obtain this feedback without the error listing file by using pipes to the calling application.

Example

```
-WOutFileOn
```

Error file is created as specified with ERRORFILE

```
-WOutFileOff
```

No error file created

See also

-WErrFile: Create "err.log" Error File

-WStdout: Write to Standard Output

7.2.3.1.113 -Wpd: Error for Implicit Parameter Declaration

Group

MESSAGES

Scope

Function

Syntax

```
-Wpd
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prompts the Compiler to issue an *ERROR* message instead of a *WARNING* message when the Compiler encounters an implicit declaration. This occurs if the Compiler does not have a prototype for the called function.

This option helps prevent parameter-passing errors, which can only be detected at runtime. It requires prototyping each called function before use. Correct ANSI behavior assumes that parameters are correct for the stated call.

This option is the same as using `-WmsgSe1801`.

Example

```
-Wpd


main() {


  char a, b;


  myfunc(a, b); // <- Error here


}


myfunc(a, b, c)


  char a, b, c;


{


  ...


}
```

See also

Message C1801

-WmsgSe: Setting a Message to Error

7.2.3.1.114 -WStdout: Write to Standard Output

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-WStdout(On|Off)
```

Arguments

None

Default

Output is written to `stdout`

Defines

None

Pragmas

None

Description

This option enables the usual standard streams with Windows applications. Text written into the streams does not appear anywhere unless explicitly requested by the calling application. This option determines whether error file text written to the error file is also written into the `stdout` file.

Example

```
-WStdoutOn
```

All messages written to stdout

```
-WErrFileOff
```

Nothing written to stdout

See also

-WErrFile: Create "err.log" Error File

-WOutFile: Create Error Listing File

7.2.3.1.115 -W1: Don't Print Information Messages

Group

MESSAGES

Scope

Function

Syntax

```
-W1
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Inhibits INFORMATION message reporting. Only WARNING and ERROR messages are generated.

Example

```
-W1
```

See also

-WmsgNi: Maximum Number of Information Messages (enter <number>)

7.2.3.1.116 -W2: Do not Print INFORMATION or WARNING Messages

Group

MESSAGES

Scope

Function

Syntax

```
-W2
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Suppresses all messages of type INFORMATION and WARNING. Only ERROR messages are generated.

Example

```
-W2
```

See also

-WmsgNi: Maximum Number of Information Messages (enter <number>)

-WmsgNw: Maximum Number of Warning Messages (enter <number>)

Chapter 8
Compiler Predefined Macros

The C-language ANSI standard requires the Compiler to predefine some macros. The Compiler provides the predefined macros listed in the following table.

Table 8-1. Macros Defined by the Compiler

Macro	Description
__LINE__	Line number in the current source file
__FILE__	Name of the source file where it appears
__DATE__	The date of compilation as a string
__TIME__	The time of compilation as a string
__STDC__	Set to 1 if the -Ansi: Strict ANSI compiler option is given. Otherwise, accept additional keywords (not ANSI standard).

The following tables lists all Compiler defines with their associated names and options.

NOTE

If these macros have no value, the Compiler treats them as if they had been defined as shown: `#define __HIWARE__`

It is also possible to log all predefined Compiler defines to a file using the -Ldf: Log Predefined Defines to File compiler option.

This chapter covers the following topics:

- Compiler Vendor Defines
- Product Defines
- Data Allocation Defines
- Defines for Compiler Option Settings
- Option Checking in C Code
- ANSI-C Standard Types size_t, wchar_t, and ptrdiff_t Defines
- Object-File Format Defines
- Bitfield Defines

8.1 Compiler Vendor Defines

The following table show s the defines identifying the Compiler vendor. Compilers in the USA may also be sold by ARCHIMEDES.

Table 8-2. Compiler Vendor Identification Defines

Name	Defined
__HIWARE__	always
__MWERKS__	always, set to 1

8.2 Product Defines

The following table shows the defines identifying the Compiler. The Compiler is a HI-CROSS+ Compiler (V5.0.x).

Table 8-3. Compiler Identification Defines

Name	Defined
__PRODUCT_HICROSS_PLUS__	Defined for V5.0 Compilers
__DEMO_MODE__	Defined if the Compiler is running in demo mode
__VERSION__	Defined and contains the version number, e.g., it is set to 5013 for a Compiler V5.0.13, or set to 3140 for a Compiler V3.1.40

8.3 Data Allocation Defines

The Compiler provides two macros that define data organization in memory: Little Endian (least significant byte first in memory) or Big Endian (most significant byte first in memory).

The Compiler provides the data a llocation macros listed in the following table.

Table 8-4. Compiler Macros Defining Little or Big Endian

Name	Defined
__LITTLE_ENDIAN__	Defined if the Compiler allocates in Little Endian order
__BIG_ENDIAN__	Defined if the Compiler allocates in Big Endian order

The following example illustrates the difference between little and big endian.

Listing: Little vs. Big Endian

```
unsigned long L  = 0x87654321;
unsigned short s = *(unsigned short*)&L; // BE: 0x8765,LE: 0x4321

unsigned char c  = *(unsigned char*)&L; // BE: 0x87,  LE: 0x21
```

8.4 Defines for Compiler Option Settings

The following t able lists defines for miscellaneous compiler option settings.

Table 8-5. Defines for Miscellaneous Compiler Option Settings

Name	Defined
__STDC__	-Ansi
__TRIGRAPHS__	-Ci
__CNI__	-Cni
__OPTIMIZE_FOR_TIME__	-Ot
__OPTIMIZE_FOR_SIZE__	-Os

8.5 Option Checking in C Code

You can also check the source to determine if an option is active. The EBNF syntax is:

```
OptionActive = __OPTION_ACTIVE__ ("string")
```

The above is used in the preprocessor and in C code, as shown:

Listing: Using __OPTION__ to Check for Active Options

```
#if __OPTION_ACTIVE__("-W2")
  // option -W2 is set

#endif

void main(void) {
```

```
  int i;

  if (__OPTION_ACTIVE__("-or")) {

    i=2;

  }

}
```

You can check all valid preprocessor options (e.g., options given at the command line, via the `default.env` or `project.ini` files, but not options added with the #pragma OPTION: Additional Options). You perform the same check in C code using `-Odocf` and `#pragma OPTION`.

As a parameter, only the option itself is tested and not a specific argument of an option.

See the following listing for a valid and an invalid use of `__OPTION_ACTIVE__`.

Listing: Using __OPTION_ACTIVE__

```
#if __OPTION_ACTIVE__("-D")    /* true if any -d option is given  */
#if __OPTION_ACTIVE__("-DABS") /* specific argument - not allowed */
```

To check for a specific define use:

```
#if defined(ABS)
```

If for some reason the Compiler cannot check the specified option (i.e., options that no longer exist), the Compiler issues the message "C1439: illegal pragma `__OPTION_ACTIVE__`".

8.6 ANSI-C Standard Types size_t, wchar_t, and ptrdiff_t Defines

ANSI provides some standard defines in `stddef.h` to deal with the implementation of defined object sizes.

The following listing shows part of the contents of `stdtypes.h` (included from `stddef.h`).

Listing: Type Definitions of ANSI-C Standard Types

```
/* size_t: defines the maximum object size type */
  #if defined(__SIZE_T_IS_UCHAR__)

    typedef unsigned char  size_t;

  #elif defined(__SIZE_T_IS_USHORT__)

    typedef unsigned short size_t;

  #elif defined(__SIZE_T_IS_UINT__)

    typedef unsigned int   size_t;
```

```
  #elif defined(__SIZE_T_IS_ULONG__)
    typedef unsigned long  size_t;
  #else
    #error "illegal size_t type"
  #endif
/* ptrdiff_t: defines the maximum pointer difference type */
  #if defined(__PTRDIFF_T_IS_CHAR__)
    typedef signed char   ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_SHORT__)
    typedef signed short  ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_INT__)
    typedef signed int    ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_LONG__)
    typedef signed long   ptrdiff_t;
  #else
    #error "illegal ptrdiff_t type"
  #endif
/* wchar_t: defines the type of wide character */
#if defined(__WCHAR_T_IS_UCHAR__)
  typedef unsigned char  wchar_t;
#elif defined(__WCHAR_T_IS_USHORT__)
  typedef unsigned short wchar_t;
#elif defined(__WCHAR_T_IS_UINT__)
  typedef unsigned int   wchar_t;
#elif defined(__WCHAR_T_IS_ULONG__)
  typedef unsigned long  wchar_t;
#else
  #error "illegal wchar_t type"
#endif
```

The following table lists define s that deal with other possible implementations:

Table 8-6. Defines for Other Implementations

Macro	Description
__SIZE_T_IS_UCHAR__	Defined if the Compiler expects `size_t` in `stddef.h` to be *unsigned char.*

Table continues on the next page...

Table 8-6. Defines for Other Implementations (continued)

Macro	Description
__SIZE_T_IS_USHORT__	Defined if the Compiler expects `size_t` in `stddef.h` to be *unsigned short.*
__SIZE_T_IS_UINT__	Defined if the Compiler expects `size_t` in `stddef.h` to be *unsigned int*.
__SIZE_T_IS_ULONG__	Defined if the Compiler expects `size_t` in `stddef.h` to be *unsigned long*.
__WCHAR_T_IS_UCHAR__	Defined if the Compiler expects `wchar_t` in `stddef.h` to be *unsignedchar*.
__WCHAR_T_IS_USHORT__	Defined if the Compiler expects `wchar_t` in `stddef.h` to be *unsignedshort*.
__WCHAR_T_IS_UINT__	Defined if the Compiler expects `wchar_t` in `stddef.h` to be *unsignedint*.
__WCHAR_T_IS_ULONG__	Defined if the Compiler expects `wchar_t` in `stddef.h` to be *unsignedlong*.
__PTRDIFF_T_IS_CHAR__	Defined if the Compiler expects `ptrdiff_t` in `stddef.h` to be *char*.
__PTRDIFF_T_IS_SHORT__	Defined if the Compiler expects `ptrdiff_t` in `stddef.h` to be *short*.
__PTRDIFF_T_IS_INT__	Defined if the Compiler expects `ptrdiff_t` in `stddef.h` to be *int*.
__PTRDIFF_T_IS_LONG__	Defined if the Compiler expects `ptrdiff_t` in `stddef.h` to be *long*.

The following tables show the default settings of the ANSI-C Compiler standard types `size_t` and `ptrdiff_t`.

This section covers the following topics:

- Macros for HC08
- Division and Modulus

8.6.1 Macros for HC08

The following table lists the `size_t` macro settings for the HC08 target.

Table 8-7. size_t Macro Settings

size_t Macro	Defined
__SIZE_T_IS_UCHAR__	-Mt
__SIZE_T_IS_USHORT__	Never
__SIZE_T_IS_UINT__	-Ms, -Mb
__SIZE_T_IS_ULONG__	Never

The following table lists the `ptrdiff_t` macro settings for the HC08 target.

Table 8-8. ptrdiff_t Macro Settings

ptrdiff_t Macro	Defined
__PTRDIFF_T_IS_CHAR__	-Mt
__PTRDIFF_T_IS_SHORT__	Never
__PTRDIFF_T_IS_INT__	-Ms
__PTRDIFF_T_IS_LONG__	Never

8.6.2 Division and Modulus

Division and Modulus

To ensure that the results of the "/" and "%" operators are defined correctly for signed arithmetic operations, both operands must be defined positive. (Refer to HC(S)08 Backend.) When one of the operands is defined negative, the implementation defines the sign of the result. See the following listing.

Listing: Effect of Polarity on Division and Modulus Arithmetic

```
#ifdef __MODULO_IS_POSITIV__
 22 / 7 ==  3;   22 %  7 == 1

 22 /-7 == -3;   22 % -7 == 1

-22 / 7 == -4;  -22 %  7 == 6

-22 /-7 ==  4;  -22 % -7 == 6

#else

 22 / 7 ==  3;   22 %  7 == +1

 22 /-7 == -3;   22 % -7 == +1

-22 / 7 == -3;  -22 %  7 == -1

-22 /-7 ==  3;  -22 % -7 == -1

#endif
```

The following table lists the modulus macro for the HC08 target.

Table 8-9. Modulus Macro Settings

Name	Defined
__MODULO_IS_POSITIV__	Never

8.7 Object-File Format Defines

The Compiler defines some macros to identify the object-file format. These macros appear primarily in object file-spec ific startup code. The following table lists these defines.

Table 8-10. Object-File Format Defines

Name	Defined
__HIWARE_OBJECT_FILE_FORMAT__	-Fh
__ELF_OBJECT_FILE_FORMAT__	-F1, -F2

8.8 Bitfield Defines

This topic explains following:

- Bitfield Allocation
- Bitfield Type Reduction
- Sign of Plain Bitfields
- Macros for HC08
- Type Information Defines
- HC08-Specific Defines

8.8.1 Bitfield Allocation

The Compiler provides six predefined bitfield allocation macros (as shown in the following listing):

Listing: Predefined Bitfield Allocation Macros

```
__BITFIELD_MSBIT_FIRST__  /* defined if bitfield allocation
                             starts with MSBit */

__BITFIELD_LSBIT_FIRST__  /* defined if bitfield allocation

                             starts with LSBit */
```

```
__BITFIELD_MSBYTE_FIRST__ /* allocation of bytes starts with
                             MSByte */

__BITFIELD_LSBYTE_FIRST__ /* allocation of bytes starts with
                             LSByte */

__BITFIELD_MSWORD_FIRST__ /* defined if bitfield allocation
                             starts with MSWord */

__BITFIELD_LSWORD_FIRST__ /* defined if bitfield allocation
                             starts with LSWord */
```

Using the defines listed above, you can write compatible code over different Compiler vendors even if the bitfield allocation differs. Note that the allocation order of bitfields is important (as shown in the following listing).

Listing: Using Predefined Bitfield Allocation Macros

```
struct {
  /* Memory layout of I/O port:
                  MSB                        LSB
      name:       BITA | CCR | DIR | DATA | DDR2
      size:       1      1     1     4      1
  */
#ifdef __BITFIELD_MSBIT_FIRST__
  unsigned int BITA:1;
  unsigned int CCR :1;
  unsigned int DIR :1;
  unsigned int DATA:4;
  unsigned int DDR2:1;
#elif defined(__BITFIELD_LSBIT_FIRST__)
  unsigned int DDR2:1;
  unsigned int DATA:4;
  unsigned int DIR :1;
  unsigned int CCR :1;
  unsigned int BITA:1;
#else
  #error "undefined bitfield allocation strategy!"
#endif
  } MyIOport;
```

If the basic bitfield allocation unit in the Compiler is a byte, the bitfield memory allocation is always from the most significant BYTE to the least significant BYTE (big endian). For example, `__BITFIELD_MSBYTE_FIRST__` is defined as shown in the following listing.

Listing: __BITFIELD_MSBYTE_FIRST__

```
struct {
  unsigned char a:8;

  unsigned char b:3;

  unsigned char c:5;

} MyIOport2;

/* LSBIT_FIRST       */  /* MSBIT_FIRST       */

/* MSByte   LSByte   */  /* MSByte   LSByte   */

/* aaaaaaaa cccccbbb */  /* aaaaaaaa bbbccccc */
```

NOTE

There is no standard way to allocate bitfields. Allocation may vary from compiler to compiler even for the same target. Using bitfields for I/O register access is non-portable and inefficient due to the masking involved in unpacking individual fields. We recommend that you use regular bit-and (`&`) and bit-or (`|`) operations for I/O port access.

8.8.2 Bitfield Type Reduction

The Compiler provides two predefined macros for enabled/disabled type-size reduction (as shown in the following listing). With type-size red uction enabled, the Compiler is free to reduce the type of a bitfield. For example, if the size of a bitfield is 3, the Compiler uses the `char` type.

Listing: Bitfield Type-Reduction Macros

```
__BITFIELD_TYPE_SIZE_REDUCTION__     /* defined if type-size
                                        reduction is enabled  */


__BITFIELD_NO_TYPE_SIZE_REDUCTION__ /* defined if type-size

                                        reduction is disabled */
```

It is possible to write compatible code over different Compiler vendors and get optimized bitfields (as shown in the following listing).

Listing: Effects of Bitfield Type-Size Reduction

```
struct{
  long b1:4;

  long b2:4;

} myBitfield;

31                        7  3  0

--------------------------------

|########################|b2|b1|
-BfaTSRoff

--------------------------------

7   3   0

---------

|b2 |b1 |
-BfaTSRon

---------
```

8.8.3 Sign of Plain Bitfields

For some architectures, the sign of a plain bitfield does not follow standard rules. Normally for the bitfield in the following listing, `myBits` is signed, because plain `int` is also signed.

Listing: Signed Bitfield

```
struct _bits {
  int myBits:3;

} bits;
```

To implement this bitfield as an unsigned bitfield, use the code in the following listing.

Listing: Unsigned Bitfield

```
struct _bits {
  unsigned int myBits:3;

} bits;
```

However, some architectures must overwrite this behavior for Embedded Application Binary Interface (EABI) compliance. Under those circumstances, the Compiler uses the *-T: Flexible Type Management* option, if supported. The option affects the following defines (as shown in the following listing):

Listing: Defines Affected by -T Option Use

```
__PLAIN_BITFIELD_IS_SIGNED__     /* defined if plain bitfield is signed */

__PLAIN_BITFIELD_IS_UNSIGNED__   /* defined if plain bitfield is unsigned */
```

8.8.4 Macros for HC08

The following table identifies the implementation in the Backend for the HC08 target.

Table 8-11. Macros for HC08

Name	Defined
__BITFIELD_MSBIT_FIRST__	-BfaBMS
__BITFIELD_LSBIT_FIRST__	-BfaBLS
__BITFIELD_MSBYTE_FIRST__	always
__BITFIELD_LSBYTE_FIRST__	never
__BITFIELD_MSWORD_FIRST__	always
__BITFIELD_LSWORD_FIRST__	never
__BITFIELD_TYPE_SIZE_REDUCTION__	-BfaTSRon
__BITFIELD_NO_TYPE_SIZE_REDUCTION__	-BfaTSRoff
__PLAIN_BITFIELD_IS_SIGNED__	Always
__PLAIN_BITFIELD_IS_UNSIGNED__	Never

8.8.5 Type Information Defines

Flexible Type Management sets the defines to identify the type sizes. the following the following lists these defines.

Table 8-12. Type Information Defines

Name	Defined
__CHAR_IS_SIGNED__	See the -T option or Backend
__CHAR_IS_UNSIGNED__	See the -T option or Backend
__CHAR_IS_8BIT__	See the -T option or Backend
__CHAR_IS_16BIT__	See the -T option or Backend
__CHAR_IS_32BIT__	See the -T option or Backend
__CHAR_IS_64BIT__	See the -T option or Backend
__SHORT_IS_8BIT__	See the -T option or Backend
__SHORT_IS_16BIT__	See the -T option or Backend

Table continues on the next page...

Table 8-12. Type Information Defines (continued)

Name	Defined
__SHORT_IS_32BIT__	See the -T option or Backend
__SHORT_IS_64BIT__	See the -T option or Backend
__INT_IS_8BIT__	See the -T option or Backend
__INT_IS_16BIT__	See the -T option or Backend
__INT_IS_32BIT__	See the -T option or Backend
__INT_IS_64BIT__	See the -T option or Backend
__ENUM_IS_8BIT__	See the -T option or Backend
__ENUM_IS_SIGNED__	See the -T option or Backend
__ENUM_IS_UNSIGNED__	See the -T option or Backend
__ENUM_IS_16BIT__	See the -T option or Backend
__ENUM_IS_32BIT__	See the -T option or Backend
__ENUM_IS_64BIT__	See the -T option or Backend
__LONG_IS_8BIT__	See the -T option or Backend
__LONG_IS_16BIT__	See the -T option or Backend
__LONG_IS_32BIT__	See the -T option or Backend
__LONG_IS_64BIT__	See the -T option or Backend
__LONG_LONG_IS_8BIT__	See the -T option or Backend
__LONG_LONG_IS_16BIT__	See the -T option or Backend
__LONG_LONG_IS_32BIT__	See the -T option or Backend
__LONG_LONG_IS_64BIT__	See the -T option or Backend
__FLOAT_IS_IEEE32__	See the -T option or Backend
__FLOAT_IS_IEEE64__	See the -T option or Backend
__FLOAT_IS_DSP__	See the -T option or Backend
__DOUBLE_IS_IEEE32__	See the -T option or Backend
__DOUBLE_IS_IEEE64__	See the -T option or Backend
__DOUBLE_IS_DSP__	See the -T option or Backend
__LONG_DOUBLE_IS_IEEE32__	See the -T option or Backend
__LONG_DOUBLE_IS_IEEE64__	See the -T option or Backend
__LONG_DOUBLE_IS_DSP__	See the -T option or Backend
__LONG_LONG_DOUBLE_IS_IEEE32__	See the -T option or Backend
__LONG_LONG_DOUBLE_IS_IEEE64__	See the -T option or Backend
__LONG_LONG_DOUBLE_IS_DSP__	See the -T option or Backend
__VTAB_DELTA_IS_8BIT__	See the -T option
__VTAB_DELTA_IS_16BIT__	See the -T option
__VTAB_DELTA_IS_32BIT__	See the -T option
__VTAB_DELTA_IS_64BIT__	See the -T option
__PLAIN_BITFIELD_IS_SIGNED__	See the -T option or Backend
__PLAIN_BITFIELD_IS_UNSIGNED__	See the -T option or Backend

8.8.6 HC08-Specific Defines

The following table identifies specific implementations in the Backend for the HC08 target.

Table 8-13. HC08-Specific Defines

Name	Defined
__HCS08__	Always
__HCS08__	-Cs08
__NO_RECURSION__	Never
__BANKED__	-Mb
__SMALL__	-Ms
__TINY__	-Mt
__MMU__	-MMU
__PTR_SIZE_1__	-Mt
__PTR_SIZE_2__	-Ms
__PTR_SIZE_3__	Never
__PTR_SIZE_4__	Never

Chapter 9
Compiler Pragmas

A pragma, as shown in the following listing, defines information that passes from the Compiler Frontend to the Compiler Backend, without affecting the parser. In the Compiler, the effect of a pragma on code generation starts at the point of its definition and ends with the end of the next function. Exceptions to this rule are the pragmas `ONCE` and `NO_STRING_CONSTR` (see #pragma ONCE: Include Once and #pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing), which are valid for only one file.

Listing: Syntax of a Pragma

```
#pragma pragma_name [optional_arguments]
```

The `optional_arguments` value depends on the pragma that you use. Some pragmas do not take arguments.

NOTE

A pragma directive accepts a single pragma with optional arguments. Do not place more than one pragma name in a pragma directive. The following example uses incorrect syntax:

```
#pragma ONCE NO_STRING_CONSTR
```

The following section describes all pragmas that affect the frontend. All other pragmas affect only the code generation process and are described in the back-end section. The topics included are as follows:

- Pragma Details

9.1 Pragma Details

This section describes each Compiler-available pragma. The pragmas are listed in alphabetical order and are divided into separate tables. The following table lists and defines the topics that appear in the description of each pragma.

Table 9-1. Pragma Documentation Topics

Topic	Description
Scope	Defines scope of pragma in which it is valid (refer to the table "*Definition of Items that can Appear in a Pragma's Scope Topic*").
Syntax	Specifies pragma syntax in an EBNF format.
Synonym	Lists a synonym for the pragma or none, if no synonym exists.
Arguments	Describes and lists optional and required arguments for the pragma.
Default	Shows the default setting for the pragma or none.
Description	Provides a detailed description of the pragma and its use.
Example	Gives an example of usage and effects of the pragma.
See also	Names related sections.

The following table describes the different scopes of pragmas.

Table 9-2. Definition of Items that can Appear in a Pragma's Scope Topic

Scope	Description
File	The pragma is valid from the current position until the end of the file. Example: If the pragma is in a header file included from a source file, the pragma is valid in the header file but not the source file.
Compilation Unit	The pragma is valid from the current position until the end of the whole compilation unit. Example: If the pragma is in a header file included from a source file, it is valid in the header file and the source file.
Data Definition	The pragma affects only the next data definition. Ensure that you always use a data definition behind this pragma in a header file. If not, the pragma is used for the first data segment in the next header file, or in the main file.
Function Definition	The pragma affects only the next function definition. Ensure that you use this pragma in a header file: The pragma is valid for the first function in each included source file if there is no function definition in the header file.
Next pragma with same name	The pragma is used until the same pragma appears again. If no such pragma follows this one, it is valid until the end of the file.

The following listed are the pragmas for HCS08 compiler:

Table 9-3. Pragmas for HCS08 Compiler

#pragma CODE_SEG: Code Segment Definition

Table continues on the next page...

Table 9-3. Pragmas for HCS08 Compiler (continued)

#pragma CONST_SEG: Constant Data Segment Definition
#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing
#pragma DATA_SEG: Data Segment Definition
#pragma DATA_SEG: Data Segment Definition
#pragma INTO_ROM: Put Next Variable Definition into ROM
#pragma LINK_INFO: Pass Information to the Linker
#pragma LOOP_UNROLL: Force Loop Unrolling
#pragma mark: Entry in CodeWarrior IDE Function List
#pragma MESSAGE: Message Setting
#pragma NO_ENTRY: No Entry Code
#pragma NO_EXIT: No Exit Code
#pragma NO_FRAME: No Frame Code
#pragma NO_INLINE: Do not Inline Next Function Definition
#pragma NO_LOOP_UNROLL: Disable Loop Unrolling
#pragma NO_RETURN: No Return Instruction
#pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing
#pragma ONCE: Include Once
#pragma OPTION: Additional Options
#pragma STRING_SEG: String Segment Definition
#pragma TEST_CODE: Check Generated Code
#pragma TRAP_PROC: Mark Function as Interrupt Function

9.1.1 #pragma CODE_SEG: Code Segment Definition

Scope

Next pragma `CODE_SEG`

Syntax

```
#pragma CODE_SEG (<Modif> <Name>|DEFAULT)
```

Synonym

```
CODE_SECTION
```

Arguments

`<Modif>`: Some of the following strings may be used:

__DIRECT_SEG (compatibility alias: DIRECT)

__NEAR_CODE

__NEAR_SEG (compatibility alias: NEAR)

__CODE_SEG (compatibility alias: CODE)

__ FAR_CODE

__FAR_SEG (compatibility alias: FAR)

NOTE

Do not use the compatibility alias in new code. It exists only for backward compatibility. Some compatibility alias names conflict with defines found in certain header files, and using compatibility alias names can cause hard to detect problems. Avoid using compatibility alias names.

The meaning of these segment modifiers are backend-dependent. Refer to HC(S)08 Backend for information on supported modifiers and their definitions.

<Name>: The segment name. You must use the segment name in the link parameter file on the left side of the assignment in the PLACEMENT section. Refer to the Linker manual for details.

Default

DEFAULT

Description

This pragma specifies the allocated function segment. The segment modifiers also specify the function's calling convention. The CODE_SEG pragma sets the current code segment. This segment places all new function definitions. Also, all function declarations, when they occur, get the current code segment.

The CODE_SEG pragma affects function declarations as well as definitions. Ensure that all function declarations and their definitions are in the same segment.

The synonym CODE_SECTION means exactly the same as CODE_SEG. See the following listing for some CODE_SEG examples.

Listing: CODE_SEG examples

```
/* in a header file */
#pragma CODE_SEG __FAR_SEG MY_CODE1

extern void f(void);

#pragma CODE_SEG MY_CODE2
```

```
extern void h(void);

#pragma CODE_SEG DEFAULT

/* in the corresponding C file: */

#pragma CODE_SEG __FAR_SEG MY_CODE1

void f(void){ /* f has FAR calling convention */

  h(); /* calls h with default calling convention */

}

#pragma CODE_SEG MY_CODE2

void h(void){ /* f has default calling convention */

  f(); /* calls f with the FAR calling convention */

}

#pragma CODE_SEG DEFAULT
```

NOTE

Not all compiler backends support a FAR calling convention.

NOTE

The calling convention can also be specified with a supported keyword. The default calling convention is chosen with the memory model.

The following listing shows some errors when using pragmas.

Listing: Improper Pragma Usage

```
#pragma DATA_SEG DATA1
#pragma CODE_SEG DATA1

/* error: segment name has different types! */

#pragma CODE_SEG DATA1

#pragma CODE_SEG __FAR_SEG DATA1

/* error: segment name has modifiers! */

#pragma CODE_SEG DATA1

void g(void);

#pragma CODE_SEG DEFAULT

void g(void) {}

/* error: g is declared in two different segments */

#pragma CODE_SEG __FAR_SEG DEFAULT

/* error: modifiers for DEFAULT segment are not allowed */
```

See also

HC(S)08 Backend

Segmentation

Linker Manual

#pragma CONST_SEG: Constant Data Segment Definition

#pragma DATA_SEG: Data Segment Definition

#pragma STRING_SEG: String Segment Definition

-Cc: Allocate Const Objects into ROM compiler option

9.1.2 #pragma CONST_SEG: Constant Data Segment Definition

Scope

Next pragma `CONST_SEG`

Syntax

```
#pragma CONST_SEG (<Modif> <Name>|DEFAULT)
```

Synonym

`CONST_SECTION`

Arguments

`Modif`

Some of the following strings may be used:

`__SHORT_SEG` (compatibility alias: `SHORT`)

`__DIRECT_SEG` (compatibility alias: `DIRECT`)

`__NEAR_SEG` (compatibility alias: `NEAR`)

`__CODE_SEG` (compatibility alias: `CODE`)

`__FAR_SEG` (compatibility alias: `FAR`)

`__LINEAR_SEG`

NOTE

Do not use a compatibility alias in new code. Aliases exist for backwards compatibility purposes only.

The segment modifiers are backend-dependent. Refer to HC(S)08 Backend to find the supported modifiers and their meanings. The __SHORT_SEG modifier specifies a segment accessed with 8-bit addresses.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Refer to the linker manual for details.

Default

DEFAULT

Description

This pragma allocates constant variables into the current data segment. The Compiler then allocates the segment to specific addresses in the link parameter file. The pragma CONST_SEG sets the current const segment. This segment places all constant variable declarations. Set the segment to default with:

```
#pragma CONST_SEG DEFAULT
```

#pragma DATA_SEG defines the current data segment in the HIWARE object-file format, unless the -Cc option is specified, and until the first #pragma CONST_SEG occurs in the source (see #pragma DATA_SEG: Data Segment Definition and -Cc: Allocate Const Objects into ROM). The-Cc option always allocates constants in constant segments in the ELF object-file format, and after the first #pragma CONST_SEG.

The CONST_SEG pragma also affects constant variable declarations and definitions. Ensure that all constant variable declarations and definitions are in the same const segment.

Some compiler optimizations assume that objects having the same segment are placed together. Backends supporting linear data access, for example, may set the page register only once for two accesses to two different variables in the same segment. This is also the case for the DEFAULT segment. When using a paged access to variables, place one segment on one page in the link parameter file.

When #pragma INTO_ROM is active, the current const segment is not used (see #pragma INTO_ROM: Put Next Variable Definition into ROM).

The synonym CONST_SECTION means exactly the same as CONST_SEG.

Examples

The following listing shows examples of the CONST_SEG pragma.

Listing: Examples of the CONST_SEG Pragma

```
/* Use the pragmas in a header file */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short;
#pragma CONST_SEG CUSTOM_CONST_MEMORY
```

```
extern const int j_custom;
#pragma CONST_SEG DEFAULT

/* Some C file, which includes the above header file code */
void main(void) {
  int k= i; /* may use short access */
  k= j;
}

/* in the C file defining the constants : */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short=7
#pragma CONST_SEG CUSTOM_CONST_MEMORY
extern const int j_custom=8;
#pragma CONST_SEG DEFAULT
```

The following listing shows code that uses the `CONST_SEG` pragma *improperly*.

Listing: Improper use of the CONST_SEG Pragma

```
#pragma DATA_SEG CONST1
#pragma CONST_SEG CONST1 /* error: segment name has different types!*/
#pragma CONST_SEG C2
#pragma CONST_SEG __SHORT_SEG C2 // error: segment name has modifiers!
#pragma CONST_SEG CONST1
extern int i;

#pragma CONST_SEG DEFAULT
int i; /* error: i is declared in two different segments */
#pragma CONST_SEG __SHORT_SEG DEFAULT // error: modifiers for DEFAULT segment are not allowed
```

See also

HC(S)08 Backend

Segmentation

Linker Manual

#pragma CODE_SEG: Code Segment Definition

#pragma DATA_SEG: Data Segment Definition

#pragma STRING_SEG: String Segment Definition

-Cc: Allocate Const Objects into ROM compiler option

#pragma INTO_ROM: Put Next Variable Definition into ROM

9.1.3 #pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing

Scope

Next pragma `CREATE_ASM_LISTING`

Syntax

```
#pragma CREATE_ASM_LISTING (ON|OFF)
```

Synonym

None

Arguments

`ON`: Creates an assembler include file for all subsequent defines or objects

`OFF`: Does not create an assembler include file for all subsequent defines or objects

Default

```
OFF
```

Description

This pragma controls whether subsequent defines or objects are printed into the assembler include file.

A new file generates only when the `-La` compiler option is specified together with a header file containing `#pragma CREATE_ASM_LISTING ON`.

Example

```
#pragma CREATE_ASM_LISTING ON


extern int i; /* i is accessible from the asm code    */


#pragma CREATE_ASM_LISTING OFF


extern int j; /* j is only accessible from the C code */
```

See also

-La: Generate Assembler Include File

Generating Assembler Include Files (-La Compiler Option)

9.1.4 #pragma DATA_SEG: Data Segment Definition

Scope

Next pragma DATA_SEG

Syntax

```
#pragma DATA_SEG (<Modif> <Name>|DEFAULT)
```

Synonym

```
DATA_SECTION
```

Arguments

<Modif>: Some of the following strings may be used:

__SHORT_SEG (compatibility alias: SHORT)

__DIRECT_SEG (compatibility alias: DIRECT)

__NEAR_SEG (compatibility alias: NEAR)

__CODE_SEG (compatibility alias: CODE)

__FAR_SEG (compatibility alias: FAR)

NOTE

Do not use a compatibility alias in new code. It only exists for backwards compatibility.

The segment modifiers are backend-dependent. Refer to HC(S)08 Backend to find the supported modifiers and their meanings. The __SHORT_SEG modifier specifies a segment accessed with 8-bit addresses.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

Default

```
DEFAULT
```

Description

This pragma allocates variables into the current data segment. This segment is then moved to specific addresses in the link parameter file.

The DATA_SEG pragma sets the current data segment. This segment is used to place all variable declarations. Set the default segment with:

```
#pragma DATA_SEG DEFAULT
```

Constants are also allocated in the current data segment in the HIWARE object-file format when the option -cc is not specified and no #pragma CONST_SEG occurred in the source. When using the -Cc: Allocate Const Objects into ROM compiler option and the ELF object-file format, constants are not allocated in the data segment.

The DATA_SEG pragma also affects data declarations, as well as definitions. Ensure that all variable declarations and definitions are in the same segment.

Some compiler optimizations assume that objects having the same segment are together. Backends supporting banked data, for example, may set the page register only once if two accesses to two different variables in the same segment are done. This is also the case for the DEFAULT segment. When using a paged access to constant variables, put one segment on one page in the link parameter file.

When #pragma INTO_ROM: Put Next Variable Definition into ROM is active, the current data segment is not used.

The DATA_SECTION synonym means exactly the same as DATA_SEG.

Example

The following listing is an example using the DATA_SEG pragma.

Listing: Using the DATA_SEG Pragma

```
/* in a header file */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY

extern int i_short;

#pragma DATA_SEG CUSTOM_MEMORY

extern int j_custom;

#pragma DATA_SEG DEFAULT

/* in the corresponding C file : */

#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY

int i_short;

#pragma DATA_SEG CUSTOM_MEMORY

int j_custom;
```

```
#pragma DATA_SEG DEFAULT

void main(void) {

  i = 1; /* may use short access */

  j = 5;

}
```

The following listing shows code that uses the DATA_SEG pragma *improperly*.

Listing: Improper Use of the DATA_SEG Pragma

```
#pragma DATA_SEG DATA1
#pragma CONST_SEG DATA1 /* error: segment name has different types! */

#pragma DATA_SEG DATA1

#pragma DATA_SEG __SHORT_SEG DATA1

/* error: segment name has modifiers! */

#pragma DATA_SEG DATA1

extern int i;

#pragma DATA_SEG DEFAULT

int i; /* error: i is declared in different segments */

#pragma DATA_SEG __SHORT_SEG DEFAULT

/* error: modifiers for DEFAULT segment are not allowed */
```

See also

HC(S)08 Backend

Segmentation

Linker section of the Build Tool Utilities manual

#pragma CODE_SEG: Code Segment Definition

#pragma CONST_SEG: Constant Data Segment Definition

#pragma STRING_SEG: String Segment Definition

-Cc: Allocate Const Objects into ROM compiler option

#pragma INTO_ROM: Put Next Variable Definition into ROM

9.1.5 #pragma INLINE: Inline Next Function Definition

Scope

Function Definition

Syntax

```
#pragma INLINE
```

Synonym

None

Arguments

None

Default

None

Description

This pragma directs the Compiler to inline the next function in the source.

The pragma produces the same results as using the `-Oi` compiler option.

Example

The following code demonstrates using an INLINE pragma to inline a function.

```
int i;

#pragma INLINE

static void myfun(void) {

  i = 12;

}

void main(void) {
```

```
    myfun(); // results in inlining `i = 12;'


  }
```

See also

#pragma NO_INLINE: Do not Inline Next Function Definition

-Oi: Inlining compiler option

9.1.6 #pragma INTO_ROM: Put Next Variable Definition into ROM

Scope

Data Definition

Syntax

```
#pragma INTO_ROM
```

Synonym

None

Arguments

None

Default

None

Description

This pragma forces the next (non-constant) variable definition to be `const` (together with the `-Cc` compiler option).

The pragma is active only for the next single variable definition. A following segment pragma (`CONST_SEG`, `DATA_SEG`, `CODE_SEG`) disables the pragma.

NOTE

This pragma is only useful for the HIWARE object-file format (and not for ELF/DWARF).

NOTE

This pragma forces the Compiler to recognize a non-constant (means normal `variable') object as `const`. If the variable already is declared as `const' in the source, this pragma is not needed. This pragma was introduced to cheat the constant handling of the compiler, and shall not be used any longer. It is supported for legacy reasons only. Do not use in new code

Example

The following listing is an example using the `INTO_ROM` pragma.

Listing: Using the INTO_ROM Pragma

```
#pragma INTO_ROM
char *const B[] = {"hello", "world"};

#pragma INTO_ROM

int constVariable; /* put into ROM_VAR, .rodata */

int other; /* put into default segment */

#pragma INTO_ROM

#pragma DATA_SEG MySeg /* INTO_ROM overwritten! */

int other2; /* put into MySeg */
```

See also

-Cc: Allocate Const Objects into ROM compiler option

9.1.7 #pragma LINK_INFO: Pass Information to the Linker

Scope

Function

Syntax

```
#pragma LINK_INFO NAME "CONTENT"
```

Synonym

None

Arguments

`NAME`: Identifier specific to the purpose of this `LINK_INFO`.

CONTENT: C style string containing only printable ASCII characters.

Default

None

Description

This pragma instructs the compiler to put the passed name-content pair into the ELF file. For the compiler, the used name and its content have no meaning, other than one NAME can only contain one content. However, multiple pragmas with different NAMEs are legal.

For the linker or the debugger however, the NAME might trigger some special functionality with CONTENT as an argument.

The linker collects the CONTENT for every NAME in different object files and issues a message if a different CONTENT is given for different object files.

NOTE

This pragma only works with the ELF object-file format.

Example

Apart from extended functionality implemented in the linker or debugger, this feature can also be used for user-defined link-time consistency checks:

Using the following listing code in a header file used by all compilation units, the linker will issue a message if the object files built with _DEBUG are linked with object files built without it.

Listing: Using Pragmas to Assist in Debugging

```
#ifdef _DEBUG
#pragma LINK_INFO MY_BUILD_ENV DEBUG

#else

#pragma LINK_INFO MY_BUILD_ENV NO_DEBUG

#endif
```

9.1.8 #pragma LOOP_UNROLL: Force Loop Unrolling

Scope

Function

Syntax

```
#pragma LOOP_UNROLL
```

Synonym

None

Arguments

None

Default

None

Description

This pragma performs loop unrolling for the next function. This is the same as setting the `-Cu` option for a subsequent single function.

Example

In the following example, the pragma unrolls a *for* loop.

Listing: Using the LOOP_UNROLL Pragma to Unroll a For Loop

```
#pragma LOOP_UNROLL
void F(void) {

  for (i=0; i<5; i++) { // unrolling this loop

  ...
```

See also

#pragma NO_LOOP_UNROLL: Disable Loop Unrolling

-Cu: Loop Unrolling

9.1.9 #pragma mark: Entry in CodeWarrior IDE Function List

Scope

Line

Syntax

```
#pragma mark {any text}
```

Synonym

None

Arguments

None

Default

None

Description

This pragma adds an entry into the CodeWarrior IDE function list. It also helps to introduce faster code lookups by providing a menu entry which jumps directly to a code position. `#pragma mark` inserts a separator line.

NOTE

The CodeWarrior IDE actually handles this pragma; the compiler ignores it. The CodeWarrior IDE scans opened source files for this pragma. It is not necessary to recompile a file when this pragma is changed. The IDE updates its menus instantly.

Example

In the following listing, the pragma accesses declarations and definitions.

Listing: Using the MARK Pragma

```
#pragma mark local function declarations
static void inc_counter(void);

static void inc_ref(void);

#pragma mark local variable definitions

static int counter;

static int ref;

#pragma mark -

static void inc_counter(void) {

  counter++;

}

static void inc_ref(void) {

  ref++;

}
```

9.1.10 #pragma MESSAGE: Message Setting

Scope

Compilation Unit or until the next `MESSAGE` pragma

Syntax

```
#pragma MESSAGE {(WARNING|ERROR| INFORMATION|DISABLE|DEFAULT){<CNUM>}}
```

Synonym

None

Arguments

`<CNUM>`: Number of message to be set in the C1234 format

Default

None

Description

This pragma selectively sets messages to information, warning, disable, or error.

NOTE

This pragma does not affect messages produced during preprocessing, because pragma parsing is done during normal source parsing but not during preprocessing.

NOTE

This pragma (as other pragmas) must be specified outside of the function scope. It is not possible to change a message inside a function or for a part of a function.

Example

In the following listing, parentheses ( ) were left out.

Listing: Using the MESSAGE Pragma

```
/* treat C1412: Not a function call, */
/* address of a function, as error */

#pragma MESSAGE ERROR C1412

void f(void);

void main(void) {
```

```
  f; /* () is missing, but still legal in C */
/* ERROR because of pragma MESSAGE */
}
```

See also

-WmsgSd: Setting a Message to Disable

-WmsgSe: Setting a Message to Error

-WmsgSi: Setting a Message to Information

-WmsgSw: Setting a Message to Warning

9.1.11 #pragma NO_ENTRY: No Entry Code

Scope

Function

Syntax

```
  #pragma NO_ENTRY
```

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of the entry code and is useful for inline assembler functions.

The code generated in a function with `#pragma NO_ENTRY` instructs the compiler to rely on the user's stack handling code. It is assumed that the user ensures safe stack use.

NOTE

Not all backends support this pragma. Some still generate entry code even when this pragma is specified.

Example

The following listing uses the NO_ENTRY pragma (along with others) to avoid any Compiler code generation. All code is written in inline assembler.

Listing: Blocking Compiler-Generated Function Management Instructions

```
#pragma NO_ENTRY
#pragma NO_EXIT

#pragma NO_FRAME

#pragma NO_RETURN

void Func0(void) {

  __asm {/* no code should be written by the compiler.*/

    ...

  }

}
```

See also

#pragma NO_EXIT: No Exit Code

#pragma NO_FRAME: No Frame Code

#pragma NO_RETURN: No Return Instruction

9.1.12 #pragma NO_EXIT: No Exit Code

Scope

Function

Syntax

```
#pragma NO_EXIT
```

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses generation of the exit code and is useful for inline assembler functions.

The code generated in a function with #pragma NO_EXIT instructs the compiler to rely on the user's stack handling code. It is assumed that the user ensures safe stack use.

NOTE

Not all backends support this pragma. Some still generate exit code even when this pragma is specified.

Example

The following listing uses the NO_EXIT pragma (along with others) to avoid any Compiler code generation. All code is written in inline assembler.

Listing: Blocking Compiler-Generated Function Management Instructions

```
#pragma NO_ENTRY
#pragma NO_EXIT

#pragma NO_FRAME

#pragma NO_RETURN

void Func0(void) {

  __asm {/* no code should be written by the compiler.*/

    ...

  }

}
```

See also

#pragma NO_ENTRY: No Entry Code

#pragma NO_FRAME: No Frame Code

#pragma NO_RETURN: No Return Instruction

9.1.13 #pragma NO_FRAME: No Frame Code

Scope

Function

Syntax

```
#pragma NO_FRAME
```

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of frame code and is useful for inline assembler functions.

The code generated in a function with #pragma `NO_FRAME` instructs the compiler to rely on the user's stack handling code. It is assumed that the user ensures safe stack use.

NOTE

Not all backends support this pragma. Some still generate frame code even when this pragma is specified.

Example

The following listing uses the `NO_FRAME` pragma (along with others) to avoid any Compiler code generation. All code is written in inline assembler.

Listing: Blocking Compiler-Generated Function Management Instructions

```
#pragma NO_ENTRY
#pragma NO_EXIT

#pragma NO_FRAME

#pragma NO_RETURN

void Func0(void) {

  __asm {/* no code should be written by the compiler.*/

    ...

  }

}
```

See also

#pragma NO_ENTRY: No Entry Code

#pragma NO_EXIT: No Exit Code

#pragma NO_RETURN: No Return Instruction

9.1.14 #pragma NO_INLINE: Do not Inline Next Function Definition

Scope

Function

Syntax

```
#pragma NO_INLINE
```

Synonym

None

Arguments

None

Default

None

Description

This pragma prevents the Compiler from inlining the next function in the source. Use this pragma to avoid inlining a function normally inlined by the `-Oi` option.

Example

Listing: Using the NO_INLINE Pragma to Prevent Function Inlining

```
// (With option -Oi)
int i;

#pragma NO_INLINE

static void myfun(void) {

  i = 12;

}

void main(void) {

  myfun(); // call is not inlined

}
```

See also

#pragma INLINE: Inline Next Function Definition

-Oi: Inlining compiler option

9.1.15 #pragma NO_LOOP_UNROLL: Disable Loop Unrolling

Scope

Function

Syntax

```
#pragma NO_LOOP_UNROLL
```

Synonym

None

Arguments

None

Default

None

Description

Using this pragma prevents loop unrolling for the next function definition (as shown in the following listing), even when the `-Cu` command line option is given.

Example

Listing: Using the NO_LOOP_UNROLL Pragma to Temporarily Halt Loop Unrolling

```
#pragma NO_LOOP_UNROLL
void F(void) {

  for (i=0; i<5; i++) { // loop is NOT unrolled

  ...

  }
```

See also

#pragma LOOP_UNROLL: Force Loop Unrolling

-Oi: Inlining compiler option

9.1.16 #pragma NO_RETURN: No Return Instruction

Scope

Function

Syntax

```
#pragma NO_RETURN
```

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of the return instruction (return from subroutine or return from interrupt). Use this pragma when you care about the return instruction itself or when you want the code to fall through to the first instruction of the next function.

This pragma does not suppress the generation of the exit code (e.g., deallocation of local variables or compiler-generated local variables). The pragma suppresses the generation of the return instruction.

NOTE

To use this feature to fall through to the next function, you must switch off smart linking in the Linker, because the next function may be not referenced from somewhere else. In addition, ensure that both functions are in a linear segment. To be on the safe side, allocate both functions into a segment that only has a linear memory area.

Example

The following listing places some functions into a special named segment. An operating system calls each function two seconds after calling the previous function. Using this pragma, functions do not return, but fall directly through to the next function, saving code size and execution time.

Listing: Blocking Compiler-Generated Function Return Instructions

```
#pragma CODE_SEG CallEvery2Secs
#pragma NO_RETURN

void Func0(void) {

  /* first function, called from OS */

  ...

} /* fall through!!!! */

#pragma NO_RETURN

void Func1(void) {

  ...

} /* fall through */

...

/* last function has to return, no pragma is used! */

void FuncLast(void) {

  ...

}
```

See also

#pragma NO_ENTRY: No Entry Code

#pragma NO_EXIT: No Exit Code

#pragma NO_FRAME: No Frame Code

9.1.17 #pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing

Scope

Compilation Unit

Syntax

```
#pragma NO_STRING_CONSTR
```

Synonym

None

Arguments

None

Default

None

Description

This pragma switches off the special handling of # as a string constructor, and is valid for the remainder of the file in which it appears. This is useful when a macro contains inline assembler statements using this character (e.g., for IMMEDIATE values).

Example

The following pseudo assembly-code macro shows the use of the pragma. Without the pragma, the Compiler handles # as a string constructor, which is undesired behavior.

Listing: Using a NO_STRING_CONSTR Pragma to Alter the Meaning of

```
#pragma NO_STRING_CONSTR
#define HALT(x)    __asm { \

                     LOAD Reg,#3 \

                     HALT x, #255\

                   }
```

See also

Using Immediate-Addressing Mode in HLI Assembler Macros

9.1.18 #pragma ONCE: Include Once

Scope

File

Syntax

```
#pragma ONCE
```

Synonym

None

Arguments

None

Default

None

Description

When this pragma appears in a header file, the Compiler opens and reads the file only once. This increases compilation speed.

Example

```
#pragma ONCE
```

See also

-Pio: Include Files Only Once compiler option

9.1.19 #pragma OPTION: Additional Options

Scope

Compilation Unit or until next pragma OPTION

Syntax

```
#pragma OPTION (ADD [<Handle>]{<Option>}|DEL ({Handle}|ALL))
```

Synonym

None

Arguments

`<Handle>`: This identifier allows selective deletion of added options.

`<Option>`: A valid option string enclosed in double quote `(")` characters

Default

None

Description

This pragma allows the addition of options inside the source code during file compilation.

NOTE

The options given on the command line or in a configuration file cannot be changed in any way.

Use the `ADD` command to add additional options to the current options. A handle may be given optionally.

Use the `DEL` command to remove all options with a specific handle or use the `ALL` keyword to remove all options.

NOTE

Only options added using #pragma `OPTION ADD` can be deleted using `#pragma OPTION DEL`.

All keywords and the handle are case-sensitive.

Restrictions:

- The -D: Macro Definition (preprocessor definition) compiler option is not allowed. Use a "`#define`" preprocessor directive instead.
- The -OdocF: Dynamic Option Configuration for Functions compiler option is not allowed. Specify this option on the command line or in a configuration file instead.
- The Message Setting compiler options have no effect:
  - -WmsgSd: Setting a Message to Disable
  - -WmsgSe: Setting a Message to Error
  - -WmsgSi: Setting a Message to Information
  - -WmsgSw: Setting a Message to Warning

  Use #pragma MESSAGE: Message Setting instead.
- Only options concerning tasks during code generation are used. Options controlling the preprocessor, for example, have no effect.
- No macros are defined for specific options.
- Only options having function scope may be used.
- The given options must not specify a conflict to any other given option.
- The pragma is not allowed inside declarations or definitions.

Example

The following example illustrates compiling a single function with the additional `-Or` option.

Listing: Using the OPTION Pragma

```
#pragma OPTION ADD function_main_handle "-Or"
int sum(int max) { /* compiled with -or */
```

```
  int i, sum=0;

  for (i = 0; i < max; i++) {

    sum += i;

  }

  return sum;

}

#pragma OPTION DEL function_main_handle

/* now the same options as before the #pragma */

/* OPTION ADD are active again */
```

The following listing shows *improper* uses of the `OPTION` pragma.

Listing: Improper Uses of the OPTION Pragma

```
#pragma OPTION ADD -Or /* ERROR, use "-Or" */
#pragma OPTION  "-Or" /* ERROR, use keyword ADD */

#pragma OPTION ADD "-Odocf=\"-Or\""

/* ERROR, "-Odocf" not allowed in this pragma */

void f(void) {

#pragma OPTION ADD "-Or"

/* ERROR, pragma not allowed inside of declarations */

}

#pragma OPTION ADD "-Cni"

#ifdef __CNI__
/* ERROR, macros are not defined for options */

/* added with the pragma */

#endif
```

9.1.20 #pragma STRING_SEG: String Segment Definition

Scope

Next pragma `STRING_SEG`

Syntax

```
  #pragma STRING_SEG (<Modif><Name>|DEFAULT)
```

Synonym

STRING_SECTION

Arguments

<Modif>: Some of the following strings may be used:

__DIRECT_SEG (compatibility alias: DIRECT)

__NEAR_SEG (compatibility alias: NEAR)

__CODE_SEG (compatibility alias: CODE)

__FAR_SEG (compatibility alias: FAR)

__LINEAR_SEG

NOTE

Do not use a compatibility alias in new code. Compatibility aliases exist for backwards compatibility only. Some compatibility alias names conflict with defines found in certain header files. Avoid using compatibility alias names.

The __SHORT_SEG modifier specifies a segment that is accessed using 8-bit addresses. The definitions of these segment modifiers are backend-dependent. Refer to HC(S)08 Backend to find the supported modifiers and their definitions.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Refer to the linker manual for details.

Default

DEFAULT

Description

This pragma allocates strings into a linker segment called STRINGS. String segments may have modifiers. The modifiers instruct the Compiler to access the strings in a special way when necessary.

The linker treats segments defined with the pragma STRING_SEG like constant segments defined with #pragma CONST_SEG and allocates the segments in ROM areas.

The pragma STRING_SEG sets the current string segment and places all new strings into the current segment.

NOTE

The linker may support an overlapping allocation of strings (e.g., the allocation of " CDE" inside of the string " ABCDE", so that both strings together need only six bytes). When putting strings into user-defined segments, the linker may no longer do this

optimization. Only use a user-defined string segment when necessary.

The synonym STRING_SECTION means exactly the same as STRING_SEG.

Example

the following shows the STRING_SEG pragma allocating strings into a segment with the name STRING_MEMORY.

Listing: Using a STRING_SEG Pragma to Allocate a Segment for Strings

```
#pragma STRING_SEG STRING_MEMORY
char* p="String1";

void f(char*);

void main(void) {

  f("String2");

}

#pragma STRING_SEG DEFAULT
```

See also

HC(S)08 Backend

Segmentation

Linker section of the Build Tool Utilities manual

#pragma CODE_SEG: Code Segment Definition

#pragma CONST_SEG: Constant Data Segment Definition

#pragma DATA_SEG: Data Segment Definition

9.1.21 #pragma TEST_CODE: Check Generated Code

Scope

Function Definition

Syntax

```
#pragma TEST_CODE CompOp <Size> {<HashCode>}

CompOp: ==|!=|<|>|<=|>=
```

Arguments

<Size>: Size of the function to be used with compare operation

<HashCode>: Optional value specifying one specific code pattern.

Default

None

Description

This pragma checks the generated code. If the check fails, the Compiler issues message C3601.

This pragma tests the following items:

- Size of the function:

  Compares the compare operator and the size given as arguments with the size of the function.

  This feature checks whether the compiler-generated code is more or less than a given boundary. To check only the hashcode, use a condition which is always TRUE, such as != 0.

- Hashcode:

  The compiler generates a 16-bit hashcode from the code of the next function. This hashcode considers:

  - The code bytes of the generated functions
  - The type, offset, and addend of any fixup.

To get the hashcode of a certain function, compile the function with an active #pragma TEST_CODE (which will fail). Then copy the computed hashcode out of the body of message C3601.

NOTE

The code generated by the compiler may change. Test failure may indicate that the test results are unclear.

Examples

The following listings present two examples of the TEST_CODE pragma.

Listing: Using TEST_CODE to Check the Size of Generated Object Code

```
/* check that an empty function is smaller */
/* than 10 bytes */

#pragma TEST_CODE < 10

void main(void) {
```

```
}
```

You can also use the TEST_CODE pragma to detect when a different code is generated, as shown in the following listing.

Listing: Using a Test_Code Pragma with a Hashcode

```
/* If the following pragma fails, check the code. */
/* If the code is OK, add the hashcode to the */

/* list of allowed codes : */

#pragma TEST_CODE != 0 25645 37594

/* check code patterns : */

/* 25645 : shift for *2 */

/* 37594 : mult for *2 */

void main(void) {

  f(2*i);

}
```

See also

Message C3601

9.1.22 #pragma TRAP_PROC: Mark Function as Interrupt Function

Scope

Function Definition

Syntax

```
#pragma TRAP_PROC
```

Arguments

See HC(S)08 Backend.

Default

None

Description

This pragma marks a function as an interrupt function. Because interrupt functions may need some special entry and exit code, you must use this pragma to mark interrupt functions.

Do not use this pragma for declarations (e.g., in header files), as the pragma is valid for the next definition.

See HC(S)08 Backend for details.

Example

The following listing marks the `MyInterrupt()` function as an interrupt function.

Listing: Using the TRAP_PROC Pragma to Mark an Interrupt Function

```
#pragma TRAP_PROC
void MyInterrupt(void) {

  ...

}
```

See also

interrupt Keyword

Chapter 10
ANSI-C Frontend

The Compiler Frontend reads the source files, does all the syntactic and semantic checking, and produces intermediate representation of the program which then is passed on to the backend to generate code. This section discusses features, restrictions, and further properties of the ANSI-C Compiler frontend.

This chapter covers the following topics:

10.1 Implementation Features

The Compiler provides a series of pragmas instead of introducing additions to the language to support features such as interrupt procedures. The Compiler implements ANSI-C according to the X3J11 standard. Refer to the reference document *"American National Standard for Programming Languages - C"*, ANSI/ISO 9899-1990.

This section covers the following topics:

10.1.1 Keywords

See the following for the complete list of ANSI-C keywords.

Listing: ANSI-C Keywords

```
auto
break
case
char

     const
continue
default
do


double
else
enum
extern


float
for       goto
if


     int
long
register
return


short
signed
sizeof
static


     struct
switch
typedef
union


unsigned
void
```

```
volatile
while
```

10.1.2 Preprocessor Directives

The Compiler supports the full set of preprocessor directives as required by the ANSI standard, as shown in the following listing.

Listing: ANSI-C Preprocessor Directives

```
#if
#ifdef
#ifndef
#else
#elif
#endif
#define
#undef
#include
#pragma
#error
#line
```

The Compiler also supports the preprocessor operators `defined`, `#`, and `##`. The special non-ANSI directive `#warning`, which is the same as #error, issues a warning message.

10.1.3 Language Extensions

The Compiler contains a language extension for ANSI-C. Use keywords to qualify pointers to distinguish them or to mark interrupt routines.

The Compiler supports the following non-ANSI compliant keywords (see HC(S)08 Backend for semantics).

The topics covered here are as follows:

- Pointer Qualifiers
- Special Keywords
- Binary Constants (0b)
- Hexadecimal Constants ($)
- The #warning Directive
- Global Variable Address Modifier (@address)
- Variable Allocation using @ "SegmentName"
- Absolute Functions
- Absolute Variables and Linking

10.1.3.1 Pointer Qualifiers

Use pointer qualifiers to distinguish different pointer types (for example, for paging). Some pointer qualifiers are also used to specify the calling convention to be used (for example, if banking is available). Use `__linear` to access data in extended memory.

Listing: Pointer Qualifiers

```
__far  (alias
far)

__near (alias
near)

__linear
```

To allow portable programming between different CPUs (or if the target CPU does not support an additional keyword), include the defines listed below in the `` `hidef.h' `` header file.

Listing: Define far and near in the hidef.h File

```
#define far   /* no far keyword supported  */
#define near  /* no near keyword supported */
```

10.1.3.2 Special Keywords

ANSI-C was not designed with embedded controllers in mind. The keywords listed in the following listing do not conform to ANSI standards, but provide a way to achieve good results from code used for embedded applications.

Listing: Special (Non-ANSI) Keywords

```
     __alignof__

     __va_sizeof__


     __interrupt (alias
interrupt)


     __asm (aliases
_asm and
asm)
```

NOTE

See the Non-ANSI Keywords *section in the HC(S)08 Backend for more details.* You can use the `__interrupt` keyword to mark

functions as interrupt functions, and to link the function to a specified interrupt vector number (not supported by all backends).

10.1.3.3 Binary Constants (0b)

Use binary notation for constants instead of hexadecimal constants or normal constants. Note that binary constants are not allowed if the -Ansi: Strict ANSI compiler option is switched on. Binary constants start with the 0b prefix, followed by a sequence of zeroes or ones.

Listing: Using Binary Constants

```
#define myBinaryConst 0b01011
  int i;

void main(void) {

  i = myBinaryConst;

}
```

10.1.3.4 Hexadecimal Constants ($)

You can use Hexadecimal constants inside High Level Inline (HLI) Assembly. For example, instead of `0x1234` you can use `$1234`.

NOTE

This is valid only for inline assembly.

10.1.3.5 The #warning Directive

The `#warning` directive is similar to the `#error` directive.

Listing: Using the #warning Directive

```
#ifndef MY_MACRO
  #warning "
MY_MACRO set to default"

  #define MY_MACRO 1234

#endif
```

10.1.3.6 Global Variable Address Modifier (@address)

Use the global variable address modifier to assign global variables to specific addresses and to access memory-mapped I/O ports. These variables, called absolute variables, have the following syntax:

```
Declaration = <TypeSpec> <Declarator>[@<Address>|@"<Section>"]
              [= <Initializer>];
```

`<TypeSpec>` is the type specifier, for example, `int`, `char`

`<Declarator>` is the identifier of the global object, for example, `i`, `glob`

`<Address>` is the absolute address of the object, for example, `0xff04`, `0x00+8`

`<Initializer>` is the value to which the global variable is initialized.

The frontend creates a segment for each global object specified with an absolute address. This address must not be inside any address range in the `SECTIONS` entries of the link parameter file, or a linker error (overlapping segments) occurs. If the specified address has a size greater than that used for addressing the default data page, pointers pointing to this global variable must be `__far`. The following listing shows an alternate way to assign global variables to specific addresses.

Listing: Assigning Global Variables to Specific Addresses

```
#pragma DATA_SEG [__SHORT_SEG] <segment_name>
```

This sets the `PLACEMENT` section in the linker parameter file. An older method of accomplishing this is shown in the following listing.

Listing: Another Method of Assigning Global Variables to Specific Addresses

```
<segment_name> INTO  READ_ONLY <Address> ;
```

Use the `LINEAR` attribute to specify linear addresses in extended memory. Since extended memory is Flash memory, use const as the global variable.

Listing: Using LINEAR to Specify an Address

```
const char const_data @ LINEAR 0x01A6AB /*0x06A6AB*/ = 0x30;
```

Listing: Using the Global Variable Address Modifier shows a correct and incorrect example of using the global variable address modifier and *Listing: Corresponding Link Parameter File Settings (PRM file)* shows a possible PRM file that corresponds with *Listing: Using the Global Variable Address Modifier.*

Listing: Using the Global Variable Address Modifier

```
int glob @0x0500 = 10; // OK, global variable "glob" is
                       // at 0x0500, initialized with 10

void g() @0x40c0;      // error (the object is a function)

void f() {

  int i @0x40cc;       // error (the object is a local variable)

}
```

Listing: Corresponding Link Parameter File Settings (PRM file)

```
/* the address 0x0500 of "glob" must not be in any address
   range of the SECTIONS entries */

SECTIONS

    MY_RAM    = READ_WRITE 0x0800 TO 0x1BFF;

    MY_ROM    = READ_ONLY  0x2000 TO 0xFEFF;

    MY_STACK  = READ_WRITE 0x1C00 TO 0x1FFF;

    MY_IO_SEG = READ_WRITE 0x0400 TO 0x4ff;

END

PLACEMENT

    IO_SEG       INTO  MY_IO_SEG;

    DEFAULT_ROM  INTO  MY_ROM;

    DEFAULT_RAM  INTO  MY_RAM;

    SSTACK       INTO  MY_STACK;

END
```

10.1.3.7 Variable Allocation using @ "SegmentName"

The following listing shows a method of directly allocating variables in a named segment, rather than using a `#pragma`.

Listing: Allocation of Variables in Named Segments

```
#pragma DATA_SEG __SHORT_SEG tiny
#pragma DATA_SEG not_tiny

#pragma DATA_SEG __SHORT_SEG tiny_b

#pragma DATA_SEG DEFAULT

int i@"tiny";

int j@"not_tiny";

int k@"tiny_b";
```

With some pragmas in a common header file and with another macro definition, you can allocate variables depending on the macro.

```
Declaration = <TypeSpec>
<Declarator>[@"<Section>"][=<Initializer>];
```

Variables declared and defined with the `@"section"` syntax behave exactly like variables declared after their respective pragmas.

- `<TypeSpec>` is the type specifier, for example, `int` or `char`
- `<Declarator>` is the identifier of your global object, for example, `i`, `glob`
- `<Section>` is the section name. Define the section name in the link parameter file as well. For example, `"MyDataSection"`.
- `<Initializer>` is the value to which the global variable is initialized.

Specify the section name using a section pragma before the declaration occurs as shown in the following listings.

Listing: Examples of Section Pragmas

```
#pragma DATA_SEC __SHORT_SEG  MY_SHORT_DATA_SEC
#pragma DATA_SEC              MY_DATA_SEC

#pragma CONST_SEC             MY_CONST_SEC

#pragma DATA_SEC   DEFAULT // not necessary, but is good practice

#pragma CONST_SEC  DEFAULT // not necessary, but is good practice

int short_var @"MY_SHORT_DATA_SEC"; // OK, accesses are short

int ext_var @"MY_DATA_SEC" = 10; // OK, goes into

                                 // MY_DATA_SECT

int def_var;  / OK, goes into DEFAULT_RAM

const int cst_var @"MY_CONST_SEC" = 10; // OK, goes into MY_CONST_SECT
```

Listing: Corresponding Link Parameter File Settings (PRM file)

```
SECTIONS
    MY_ZRAM  = READ_WRITE 0x00F0 TO 0x00FF;

    MY_RAM   = READ_WRITE 0x0100 TO 0x01FF;

    MY_ROM   = READ_ONLY  0x2000 TO 0xFEFF;

    MY_STACK = READ_WRITE 0x0200 TO 0x03FF;

END

PLACEMENT

    MY_CONST_SEC,DEFAULT_ROM INTO  MY_ROM;

    MY_SHORT_DATA_SEC        INTO  MY_ZRAM;

    MY_DATA_SEC, DEFAULT_RAM INTO  MY_RAM;
```

```
    SSTACK                  INTO  MY_STACK;

END
```

10.1.3.8 Absolute Functions

Tthe following listing demonstrates calling an absolute function (for example, a special function in ROM) using normal ANSI-C.

Listing: Absolute Function

```
#define erase ((void(*)(void))(0xfc06))
void main(void) {

  erase(); /* call function at address 0xfc06 */

}
```

10.1.3.9 Absolute Variables and Linking

Special attention is needed if absolute variables are involved in the linker's link process. Unless the absolute object is referenced by the application, the absolute object always links in ELF/DWARF format, not HIWARE format. To force linking, switch off smart linking in the Linker, or use the ENTRIES command in the linker parameter file.

NOTE

Interrupt vector entries are always linked.

The following example shows linker handling of different absolute variables.

Listing: Linker Handling of Absolute Variables

```
      char i;          /* zero out            */
      char j = 1;      /* zero out, copy-down */

const char k = 2;      /* download  */

      char I@0x10;     /* no zero out! */

      char J@0x11 = 1;/* copy down    */

const char K@0x12 = 2;/* HIWARE: copy down / ELF: download! */

static       char L@0x13;      /* no zero out! */

static       char M@0x14 = 3;  /* copy down    */

static const char N@0x15 = 4;  /* HIWARE: copy down, ELF: download */

void interrupt 2 MyISRfct(void) {} /* download, always linked! */
```

```
  /* vector number two is downloaded with &MyISRfct */
void myfun(char *p) {} /* download */
void main(void) { /* download */
  myfun(&i); myfun(&j); myfun(&k);
  myfun(&I); myfun(&J); myfun(&K);
  myfun(&L); myfun(&M); myfun(&N);
}
```

Zero out means that the default startup code initializes the variables during startup. *Copy down* means that the variable is initialized during the default startup. To *download* means that the memory is initialized while downloading the application.

10.1.4 __far Keyword

The keyword `far` is a synonym for `__far`, which is not allowed when the -Ansi: Strict ANSI compiler option is present.

NOTE

Some Backends may not support this keyword. For more information, refer to the topic Non-ANSI Keywords in HC(S)08 Backend.

A `__far` pointer allows access to the whole memory range supported by the processor, not just to the default data page. Use it to access memory-mapped I/O registers not located on the data page. You can also use it to allocate constant strings in a ROM not on the data page.

The `__far` keyword defines the calling convention for a function. Some backends support special calling conventions which also set a page register when a function is called. This enables you to use more code than the address space can usually accommodate. The special allocation of such functions is not done automatically.

The topics covered here are as follows:

- Using __far Keyword for Pointers
- __far and Arrays
- __far and typedef Names
- __far and Global Variables
- __far and C++ Classes
- __far and C++ References
- Using __far Keyword for Functions

10.1.4.1 Using __far Keyword for Pointers

The keyword __far is a type qualifier like const and is valid only in the context of pointer types and functions.The __far keyword (for pointers) always affects the last * to its left in a type definition. The declaration of a __far pointer to a __far pointer to a character is:

```
char *__far *__far p;
```

The following is a declaration of a normal (short) pointer to a __far pointer to a character:

```
char *__far * p;
```

NOTE

To declare a __far pointer, place the __ far keyword *after* the asterisk: `char *__far p;`

10.1.4.2 __far and Arrays

The __far keyword does not appear in the context of the * type constructor in the declaration of an array parameter, as shown:

```
void my_func (char a[37]);
```

Such a declaration specifies a pointer argument. This is equal to:

```
void my_func (char *a);
```

Use one of two possible methods when declaring such an argument to a __far pointer:

```
void my_func (char a[37] __far);
```

or alternately

```
void my_func (char *__far a);
```

In the context of the [ ] type constructor in a direct parameter declaration, the __far keyword always affects the first dimension of the array to its left. In the following declaration, parameter a has type "__far pointer to array of 5 __far pointers to char":

```
void my_func (char *__far a[][5] __far);
```

10.1.4.3 __far and typedef Names

If the array type has been defi ned as a `typedef` name, as in:

```
typedef int ARRAY[10];
```

then a `__far` parameter declaration is:

```
void my_func (ARRAY __far a);
```

The parameter is a `__far` pointer to the first element of the array. This is equal to:

```
void my_func (int *__far a);
```

It is also equal to the following direct declaration:

```
void my_func (int a[10] __far);
```

It is *not* the same as specifying a `__far` pointer to the array:

```
void my_func (ARRAY *__far a);
```

because `a` has type " `__far` pointer to `ARRAY`" instead of " `__far` pointer to `int`".

10.1.4.4 __far and Global Variables

You can also use the `__far` keyword for global variables:

```
int __far i;         // OK for global variables
int __far *i;        // OK for global variables
int __far *__far i; // OK for global variables
```

This forces the Compiler to address the variable as if it has been declared in a `__FAR_SEG` segment. Note that for the above variable declarations or definitions, the variables are in the `DEFAULT_DATA` segment if no other data segment is active. Be careful when mixing `__far` declarations or definitions within a non- `__FAR_SEG` data segment. Assuming that `__FAR_SEG` segments use extended addressing mode and normal segments use direct addressing

mode, the following two examples (the listings *Acceptable, Consistent Declarations* and *Mixing Extended Addressing and Direct Addressing Modes*) clarify the resulting behavior:

Listing: Acceptable, Consistent Declarations

```
#pragma DATA_SEG MyDirectSeg        // use direct addressing mode
int i;       // direct, segment MyDirectSeg

int j;       // direct, segment MyDirectSeg

#pragma DATA_SEG __FAR_SEG MyFarSeg /* use extended addressing mode */

int k;       // extended, segment MyFarSeg

int l;       // extended, segment MyFarSeg

int __far m; // extended, segment MyFarSeg
```

Listing: Mixing Extended Addressing and Direct Addressing Modes

```
// caution: not consistent!!!!
#pragma DATA_SEG MyDirectSeg  /* use direct-addressing mode */

int i;       // direct, segment MyDirectSeg

int j;       // direct, segment MyDirectSeg

int __far k; // extended, segment MyDirectSeg

int __far l; // extended, segment MyDirectSeg

int __far m; // extended, segment MyDirectSeg
```

NOTE

The `__far` keyword global variables only affect the variable addressing mode and NOT the allocation.

10.1.4.5 __far and C++ Classes

If a member function gets the modifier `__far`, the `this` pointer is a `__far` pointer in the context of a call to that function. This is useful, for instance, if the owner class of the function is not allocated on the default data page. See the following listing.

Listing: __far Member Functions

```
class A {
public:

  void f_far(void) __far {

    /* __far version of member function A::f() */

  }

  void f(void) {
```

```
    /* normal version of member function A::f() */
  }
};
#pragma DATA_SEG MyDirectSeg // use direct addressing mode
A a_normal; // normal instance
#pragma DATA_SEG __FAR_SEG MyFarSeg // use extended addressing mode
A __far a_far;   // __far instance
void main(void)
  a_normal.f();  // call normal version of A::f() for normal instance
  a_far.f_far(); // call __far version of A::f() for __far instance
}
```

With inheritance, it no longer suffices to use `__far` with member functions only. Instead, the whole class should be qualified with the `__far` modifier. Thus, the compiler is instructed to handle '`this`' as a far pointer in all the contexts related to the declaration and definition of that class, for example vptr initialization, assignment operator generation etc. See the following listing.

Listing: _ _far modifier - Inheritance

```
class B __ far
{ ... }
class A __far : B
{ ... }
#pragma push
#pragma DATA_SEG __FAR_SEG MY_FAR_RAM
A a_far;
#pragma pop
```

NOTE

In case of inheritance, one should qualify both the derived and the base class. If the modifier has been used for the derived class only, the compiler will report the following warning:

10.1.4.6 __far and C++ References

You can apply the `__far` modifier to references. Use this option when the reference is to an object outside of the default data page. See the following listing.

Listing: __far Modifier Applied to References

```
int j; // object j allocated outside the default data page
       // (must be specified in the link parameter file)

void f(void) {

  int &__far i = j;

};
```

10.1.4.7 Using __far Keyword for Functions

This specifies a special calling convention for the `__far` keyword. Specify the `__far` keyword before the function identifier:

```
void __far f(void);
```

If the function returns a pointer, you must write the `__far` keyword before the first asterisk (" `*`").

```
int __far *f(void);
```

It must, however, be after the `int` and not before it.

For function pointers, many backends assume that the `__far` function pointer is pointing to functions with the `__far` calling convention, even when the calling convention is unspecified. Moreover, most backends do not support different function pointer sizes in one compilation unit. The function pointer size is then dependent only upon the memory model. See HC(S)08 Backend for details.

Table 10-1. Interpretation of the __far Keyword

Declaration	Allowed	Type Description
int __far f();	OK	__far function returning an int
__far int f();	error	
__far f();	OK	__far function returning an int
int __far *f();	OK	__far function returning a pointer to int
int * __far f();	OK	Function returning a __far pointer to int
__far int * f();	error	
int __far * __far f();	OK	__far function returning a __far pointer to int
int __far i;	OK	Global __far object
int __far *i;	OK	Pointer to a __far object
int * __far i;	OK	__far pointer to int

Table continues on the next page...

Table 10-1. Interpretation of the __far Keyword (continued)

Declaration	Allowed	Type Description
int __far * __far i;	OK	__far pointer to a __far object
__far int *i;	OK	Pointer to a __far integer
int *__far (* __far f)(void)	OK	__far pointer to function returning a __far pointer to int
void * __far (* f)(void)	OK	Pointer to function returning a __far pointer to void
void __far * (* f)(void)	OK	Pointer to __far function returning a pointer to void

10.1.5 __near Keyword

NOTE

See the Non-ANSI Keywords *section in* HC(S)08 Backend . The `near` keyword is a synonym for `__near`. The `near` keyword is only allowed when the -Ansi: Strict ANSI compiler option is present.

You can use the `__near` keyword instead of the `__far` keyword. Use the `__near` keyword when you must specify an explicit `__near` access and `__far` is the non-qualified pointer or when you must explicitly specify the `__near` calling convention.

The `__near` keyword uses two semantic variations. Either it specifies a small function or data pointer size or it specifies the `__near` calling convention.

Table 10-2. Interpretation of the __near Keyword

Declaration	Allowed	Type Description
`int __near f();`	OK	`__near` function returning an `int`
`int __near __far f();`	error	
`__near f();`	OK	`__near` function returning an `int`
`int __near * __far f();`	OK	`__near` function returning a `__far` pointer to `int`
`int __far *i;`	error	
`int * __near i;`	OK	`__far` pointer to `int`
`int * __far* __near i;`	OK	`__near` pointer to `__far` pointer to `int`
`int *__far (* __near f)(void)`	OK	`__near` pointer to function returning a `__far` pointer to `int`
`void * __near (* f)(void)`	OK	Pointer to function returning a `__near` pointer to `void`

Table continues on the next page...

Table 10-2. Interpretation of the __near Keyword (continued)

Declaration	Allowed	Type Description
`void __far *__near (*__near f) (void)`	OK	`__near` pointer to `__far` function returning a `__far` pointer to `void`

The topics covered here are as follows:

- Compatibility
- __alignof__ Keyword

10.1.5.1 Compatibility

`__far` pointers and normal po inters are compatible. If necessary, the normal pointer is extended to a `__far` pointer (subtraction of two pointers or assignment to a `__far` pointer). In other cases, the frontend clips the `__far` pointer to a normal pointer (that is, the page part is discarded).

10.1.5.2 __alignof__ Keyword

Some processors align objects according to their type. The unary operator, `__alignof__`, determines the alignment of a specific type. By providing any type, this operator returns its alignment. This operator behaves in the same way as `sizeof(type_name)` operator. See the target backend section to check which alignment corresponds to which fundamental data type (if any is required) or to which aggregate type (structure, array).

This macro may be useful for the `va_arg` macro in `stdarg.h`, for example, to differentiate the alignment of a structure containing four objects of four bytes from that of a structure containing two objects of eight bytes. In both cases, the size of the structure is 16 bytes, but the alignment may differ, as shown in the following listing:

Listing: va_arg Macro

```
#define va_arg(ap,type)     \
  (((__alignof__(type)>=8) ? \

    ((ap) = (char *)(((int)(ap) \

    + __alignof__(type) - 1) & (~(__alignof__(type) - 1)))) \

    : 0), \

  ((ap) += __va_rounded_size(type)),\
```

```
  (((type *) (ap))[-1]))
```

10.1.6 __va_sizeof__ Keyword

According to the ANSI-C specification, you must promote character arguments in open parameter lists to `int`. The use of `char` in the `va_arg` macro to access this parameter may not work as per the ANSI-C specification, as shown in the following listing.

Listing: Inappropriate use of char with the va_arg Macro

```
int f(int n, ...) {
  int res;

  va_list l= va_start(n, int);

  res= va_arg(l, char); /* should be va_arg(l, int) */

  va_end(l);

  return res;

}

void main(void) {

  char c=2;

  int res=f(1,c);

}
```

With the `__va_sizeof__` operator, the `f` function in the `va_arg` macro returns *2*.

A safe implementation of the `f` function is to use `va_arg(l, int)` instead of `va_arg(l, char)`.

The `__va_sizeof__` unary operator, which is used exactly as the `sizeof` keyword, returns the size of its argument after promotion as in an open parameter list, as shown in the following listing.

Listing: __va_sizeof__ Examples

```
__va_sizeof__(char) == sizeof (int)
__va_sizeof__(float) == sizeof (double)

struct A { char a; };

__va_sizeof__(struct A) >= 1 (1 if the target needs no padding bytes)
```

NOTE

In ANSI-C, it is impossible to distinguish a 1-byte structure without alignment or padding from a character variable in a `va_arg` macro. They need a different space on the open parameter calls stack for some processors.

10.1.7 interrupt Keyword

The `__interrupt` keyword is a synonym for `interrupt`, which is allowed when using the -Ansi: Strict ANSI compiler option. *See* Non-ANSI Keywords *in* HC(S)08 Backend. One of two ways can be used to specify a function as an interrupt routine:

- Use #pragma TRAP_PROC: Mark Function as Interrupt Function and adapt the Linker parameter file.
- Use the non-standard interrupt keyword.

Use the non-standard interrupt keyword like any other type qualifier (as shown in the following listing). The keyword specifies a function as an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that contains the address of the interrupt routine.

If the non-standard interrupt keyword is not followed by any number, the interrupt keyword functions the same as the `TRAP_PROC` pragma, specifying the function as an interrupt routine. However, you must associate the number of the interrupt vector with the name of the interrupt function by using the Linker's `VECTOR` directive in the Linker parameter file.

Listing: Examples of the Interrupt Keyword

```
interrupt void f(); // OK
// same as #pragma TRAP_PROC,

// please set the entry number in the prm-file

interrupt 2 int g();

// The 2nd entry (number 2) gets the address of func g().

interrupt 3 int g(); // OK

// third entry in vector points to g()

interrupt int l; // error: not a function
```

10.1.8 __asm Keyword

The Compiler supports target processor instructions inside C functions.

The `asm` keyword is a synonym for `__asm`, which is allowed when the -Ansi: Strict ANSI compiler option is not present (as shown in the following listing).

See Non-ANSI Keywords in HC(S)08 Backend for details.

Listing: Examples of the __asm Keyword

```
__asm {   nop
  nop ; comment
}
asm ("nop; nop");
  __asm("nop\n nop");
  __asm "nop";
  __asm nop;

#asm

  nop

  nop


#endasm
```

10.1.9 Intrinsic Functions

ANSI-C does not provide a mechanism to efficiently read a processor flag.

The topic covered here:

- Read Processor Flags

10.1.9.1 Read Processor Flags

To avoid using HLI for this purpose, the Compiler offers a set of intrinsic functions and inlines the function code. The associated intrinsic functions read the processor flags listed in the following listing.

Table 10-3. Read Processor Flags

Flag	Flag Abbreviation	Intrinsic Function Name
Carry	C	`__isflag_carry()`
Half carry	H	`__isflag_half_carry()`
Overflow	V	`__isflag_overflow()`
Interrupt pin high	I	`__isflag_int()`
Interrupt enable	M	`__isflag_int_enabled()`

Example:

```
if(__isflag_carry()) goto label
```

This translates to a conditional branch to `label`, that is, the process branches if the carry flag is set (for the HC08, the resulting code is BCS label).

10.1.10 Implementation-Defined Behavior

In some instances, the ANSI standard leaves the behavior of some Compilers undefined. Different Compilers may implement certain features in different ways, even if they all comply with the ANSI-C standard. The following topics discuss those points and the behavior implemented by the Compiler:

- Right Shifts
- Initialization of Aggregates with Non-Constants
- Sign of char
- Division and Modulus

10.1.10.1 Right Shifts

The result of `E1 >> E2` is implementation-defined for a right shift of an object with a signed type having a negative value if `E1` has a signed type and a negative value.

In this implementation, the Compiler performs an arithmetic right shift.

10.1.10.2 Initialization of Aggregates with Non-Constants

The initialization of aggrega tes with non-constants is not allowed in the ANSI-C specification. The Compiler allows it if the -Ansi: Strict ANSI compiler option is not set (see the following listing).

Listing: Initialization using a Non-Constant

```
void main() {
  struct A {

    struct A *n;

  } v={&v}; /* the address of v is not constant */
```

```
}
```

10.1.10.3 Sign of char

The ANSI-C standard does not specify whether the data type `char` is signed or unsigned. Refer to HC(S)08 Backend for data about default settings.

10.1.10.4 Division and Modulus

Signed arithmetic operations using the `/` and `%` operators return undefined results unless both operands are positive.

NOTE

The hardware implementation of the target's division instructions determines how a Compiler implements `/` and `%` for negative operands.

10.1.11 Translation Limitations

This section describes the internal Compiler limitations. Some stack limitations depend on the operating system used. For example, in some operating systems, limits depend on whether the compiler is a 32-bit compiler running on a 32-bit platform (for example, Windows XP), or a 16-bit Compiler running on a 16-bit platform (for example, Windows for Workgroups).

The *ANSI-C* column in the following below shows the recommended limitations of ANSI-C (5.2.4.1 in ISO/IEC 9899:1990 (E)) standard. These limitations are only guidelines and do not determine compliance. The *Implementation* column shows the actual implementation value and the possible message number. A dash in the Implementation column means that there is no information available for this topic.

Table 10-4. Translation Limitations (ANSI)

Limitation	Implementation	ANSI-C
Nesting levels of compound statements, iteration control structures, and selection control structures	256 (C1808)	15

Table continues on the next page...

Table 10-4. Translation Limitations (ANSI) (continued)

Limitation	Implementation	ANSI-C
Nesting levels of conditional inclusion	-	8
Pointer, array, and function decorators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	-	12
Nesting levels of parenthesized expressions within a full expression	32 (C4006)	32
Number of initial characters in an internal identifier or macro name	32,767	31
Number of initial characters in an external identifier	32,767	6
External identifiers in one translation unit	-	511
Identifiers with block scope declared in one block	-	127
Macro identifiers simultaneously defined in one translation unit	655,360,000 (C4403)	1024
Parameters in one function definition	-	31
Arguments in one function call	-	31
Parameters in one macro definition	1024 (C4428)	31
Arguments in one macro invocation	2048 (C4411)	31
Characters in one logical source line	2^31	509
Characters in a character string literal or wide string literal (after concatenation)	8196 (C3301, C4408, C4421)	509
Size of an object	32,767	32,767
Nesting levels for `#include` files	512 (C3000)	8
Case labels for a switch statement (excluding those for any nested switch statements)	1000	257
Data members in a single class, structure, or union	-	127
Enumeration constants in a single enumeration	-	127
Levels of nested class, structure, or union definitions in a single struct declaration list	32	15
Functions registered by `atexit()`	-	n/a
Direct and indirect base classes	-	n/a
Direct base classes for a single class	-	n/a
Members declared in a single class	-	n/a
Final overriding virtual functions in a class, accessible or not	-	n/a
Direct and indirect virtual bases of a class	-	n/a
Static members of a class	-	n/a

Table continues on the next page...

Table 10-4. Translation Limitations (ANSI) (continued)

Limitation	Implementation	ANSI-C
Friend declarations in a class	-	n/a
Access control declarations in a class	-	n/a
Member initializers in a constructor definition	-	n/a
Scope qualifications of one identifier	-	n/a
Nested external specifications	-	n/a
Template arguments in a template declaration	-	n/a
Recursively nested template instantiations	-	n/a
Handlers per try block	-	n/a
Throw specifications on a single function declaration	-	n/a

The table below shows other limitations which are not mentioned in an ANSI standard:

Table 10-5. Translation Limitations (Non-ANSI)

Limitation	Description
Type Declarations	Derived types must not contain more than 100 components.
Labels	There may be at most 16 other labels within one procedure.
Macro Expansion	Recursive macros expansion is limited to 70 (16-bit OS) or 2048 (32-bit OS) recursive expansions (C4412).
Include Files	The total number of include files is limited to 8196 for a single compilation unit.
Numbers	Maximum of 655,360,000 different numbers for a single compilation unit (C2700, C3302).
Goto	M68000 products only: Maximum of 512 Gotos for a single function (C15300).
Parsing Recursion	Maximum of 1024 parsing recursions (C2803).
Lexical Tokens	Limited by memory only (C3200).
Internal IDs	Maximum of 16,777,216 internal IDs for a single compilation unit (C3304). Internal IDs are used for additional local or global variables created by the Compiler (for example, by using CSE).
Code Size	Code size is limited to 32KB for each single function.
Filenames	Maximum length for filenames (including path) is 128 characters for 16-bit applications or 256 characters for Win32 applications. UNIX versions support filenames without the path of 64 characters in length and 256 characters with the path. Paths may be 96 characters on 16-bit PC versions, 192 characters on UNIX versions or 256 characters on 32-bit PC versions.

10.2 ANSI-C Standard

This section provides a short overview about the implementation of the ANSI-C conversion rules (see also ANSI Standard 6.x). The topics covered here are as follows:

- Integral Promotions
- Signed and Unsigned Integers
- Arithmetic Conversions
- Order of Operand Evaluation
- Rules for Standard Type Sizes

10.2.1 Integral Promotions

You may use a `char`, a `shortint`, or an `int` bitfield, or their signed or unsigned varieties, or an `enum` type, in an expression wherever an `int` or `unsignedint` is used. If an `int` represents all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsignedint`. Integral promotions preserve the value including its sign.

10.2.2 Signed and Unsigned Integers

Promoting a signed integer type to another signed integer type of greater size requires `signextension`. In two's-complement representation, the bit pattern is unchanged, except for filling the high-order bits with copies of the sign bit.

When converting a signed integer type to an unsigned inter type, if the destination has equal or greater size, the first signed extension of the signed integer type is performed. If the destination has a smaller size, the result is the remainder on division by a number, one greater than the largest unsigned number, that is represented in the type with the smaller size.

10.2.3 Arithmetic Conversions

The operands of binary operators do implicit conversions:

- If either operand has type `long double`, the other operand is converted to `long double`.
- If either operand has type `double`, the other operand is converted to `double`.
- If either operand has type `float`, the other operand is converted to `float`.
- The integral promotions are performed on both operands.

The following rules are applied:

- If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
- If one operand has type `longint` and the other has type `unsignedint`, then:
  - if a `longint` can represent all values of an `unsignedint`, the operand of type `unsignedint` is converted to `longint`;
  - if a `long int` cannot represent all the values of an `unsignedint`, both operands are converted to `unsignedlongint`.
- If either operand has type `longint`, the other operand is converted to `longint`.
- If either operand has type `unsignedint`, the other operand is converted to `unsignedint`.
- Both operands have type `int`.

10.2.4 Order of Operand Evaluation

The following listing lists the prio rity order of operators and their associativity.

Listing: Operator Precedence

```
Operators                          Associativity
() [] -> .                         left to right

! ~ ++ -- + - * & (type) sizeof    right to left

& / %                              left to right

+ -                                left to right

<< >>                              left to right

< <= > >=                          left to right

== !=                              left to right

&                                  left to right

^                                  left to right

|                                  left to right

&&                                 left to right

||                                 left to right

? :                                right to left

= += -= *= /= %= &= ^= |= <<= >>=  right to left
```

```
,                                   left to right
```

Unary `+`, `-`, and `*` have higher precedence than the binary forms.

The following listing has some examples of operator precedence

Listing: Examples of Operator Precedence

```
if (a&3 == 2)
`==' has higher precedence than `&', thus it is evaluated as:

if (a & (3==2)

which is the same as:

if (a&0)
```

Tip

Use brackets if you are unsure about associativity.

10.2.5 Rules for Standard Type Sizes

In ANSI-C, enu merations have the type of `int`. In this implementation they must be smaller than or equal to `int`. The following listing lists the size rules for integer types.

Listing: Size Relationships among Integer Types

```
sizeof(char)   <= sizeof(short)
sizeof(short)  <= sizeof(int)

sizeof(int)    <= sizeof(long)

sizeof(long)   <= sizeof(long long)

sizeof(float)  <= sizeof(double)

sizeof(double) <= sizeof(long double)
```

10.3 Floating-Point Formats

The Compiler supports two IEEE floating-point formats: IEEE32 and IEEE64. The processor may also support a DSP format. The following listing shows these three formats.

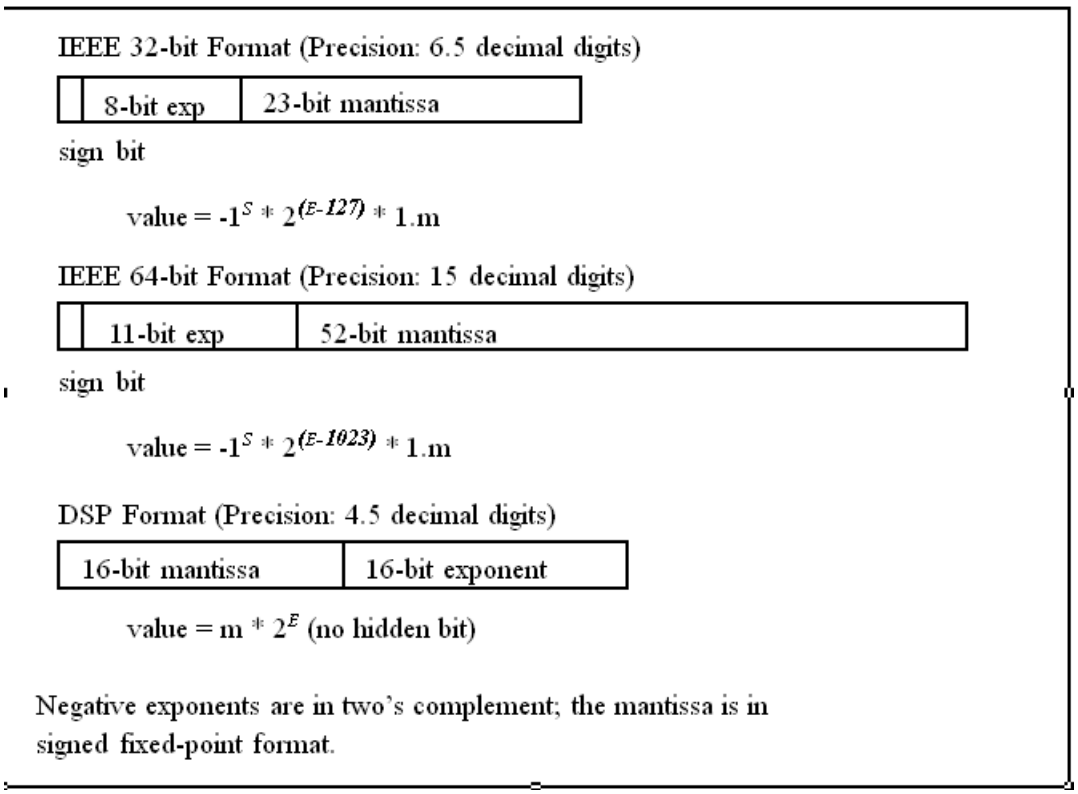

Figure 10-1. Floating-Point Formats

Floats are implemented as IEEE32 and doubles as IEEE64. This may vary for a specific Backend, or possibly neither format is supported. Refer to HC(S)08 Backend for details, default settings, and supported formats.

The topics covered here are as follows:

- Floating-Point Representation of 500.0 for IEEE
- Representation of 500.0 in IEEE32 Format
- Representation of 500.0 in IEEE64 Format
- Representation of 500.0 in DSP Format

10.3.1 Floating-Point Representation of 500.0 for IEEE

First, convert `500.0` from the decimal representation to a base 2 representation:

```
value = (-1)^s * m*2^exp
```

where: s, sign is 0 or 1,

`2 > m >= 1` for IEEE,

and exp is a integral number.

For 500, this gives:

```
sign (500.0) = 1,


m, mant (500.0, IEEE) = 1.953125, and


exp (500.0, IEEE) = 8
```

NOTE

The number 0 (zero) cannot be represented this way. For 0, IEEE defines a special bit pattern consisting of 0 bits only.

Next, convert the mantissa into its binary representation.

```
mant (500.0, IEEE) = 1.953125


= 1*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)


          + 0*2^(-5) + 1*2^(-6) + 0*....


= 1.111101000... (binary)
```

Because this number is converted to be larger or equal to 1 and smaller than 2, there is always a 1 in front of the decimal point. For the remaining steps, this constant (1) is left out in order to save space.

```
mant (500.0, IEEE, cut) = .111101000...
```

The exponent must also be converted to binary format:

```
exp (500.0, IEEE) = 8 == 08 (hex) ==  1000 (binary)
```

For the IEEE formats, the sign is encoded as a separate bit (sign magnitude representation)

10.3.2 Representation of 500.0 in IEEE32 Format

The exponent in IEEE32 has a fixed offset of 127 to ensure positive values:

```
exp (500.0, IEEE32) = 8 + 127 == 87 (hex) ==  10000111 (bin)
```

Combine the fields as shown the following listing.

Listing: Representation of Decimal 500.0 in IEEE32

```
  = 0 (sign) 10000111 (exponent)
    11110100000000000000000 (mantissa)      (IEEE32 as bin)

  = 0100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)

  = 43 fa 00 00 (IEEE32 as hex)
```

The following listing shows the IEEE32 representation of decimal -500.

Listing: Representation of Decimal -500.0 in IEEE32

```
  = 1 (sign) 10000111 (exponent)
    11110100000000000000000 (mantissa)      (IEEE32 as bin)

  = 1100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)

  = C3 fa 00 00 (IEEE32 as hex)
```

10.3.3 Representation of 500.0 in IEEE64 Format

The exponent in IEEE64 has a fixed offset of 1023 to ensure positive values:

```
exp (500.0, IEEE64) = 8 + 1023 == 407 (hex) ==  10000000111


                                                (bin)
```

The IEEE64 format is similar to IEEE32 except that more bits are available to represent the exponent and the mantissa. The following listing shows the IEEE64 representation of decimal 500.

Listing: Representation of Decimal 500.0 in IEEE64

```
  = 0 (sign) 10000000111 (exponent)
    1111010000000000000000000000000000000000000000000000 (mantissa)

                                                        (IEEE64 as bin)

  = 0100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000

         0000 0000 0000 0000 0000 (IEEE64 as bin)

  = 40 7f 40 00 00 00 00 00 (IEEE64 as hex)
```

The following listing shows the IEEE64 representation of decimal -500.

Listing: Representation of Decimal -500.0 in IEEE64

```
  = 1 (sign) 10000000111 (exponent)
    1111010000000000000000000000000000000000000000000000 (mantissa)

                                                          (IEEE64 as bin)

  = 1100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000

         0000 0000 0000 0000 0000 (IEEE64 as bin)

= c0 7f 40 00 00 00 00 00 (IEEE64 as hex)
```

NOTE

The IEEE formats recognize several special bit patterns for special values. The number 0 (zero) is encoded by the bit pattern consisting of zero bits only. Other special values such as *Not a Number*, *infinity*, *-0* (*minus zero*) and denormalized numbers exist. Refer to the IEEE standard documentation for details.

10.3.4 Representation of 500.0 in DSP Format

Convert 500.0 from the decimal representation to a representation with base 2. In contrast to IEEE, DSP normalizes the mantissa between 0 and 1 and not between 1 and 2. This makes it possible to also represent 0, which must have a special pattern in IEEE. In addition, the exponent is different from IEEE.

```
value = (-1)^s * m*2^e
```

where:

- `s`, sign, is 1 or -1,
- `1 > m >= 0, and`
- `exp` is a integral number.

For 500 this gives:

- `sign` (500.0) = 1
- `mant` (500.0, DSP) = 0.9765625
- `exp` (500.0, DSP) = 9

Next convert the mantissa into its binary representation (as shown in the following listing).

Listing: Representation of 500 in DSP format

```
mant (500.0, DSP) = 0.9765625 (dec)
  = 0*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)

            + 1*2^(-5) + 0*2^(-6) + 1*2^(-7) + 0*....

  = 0.1111101000... (bin)
```

Because this number is always greater than or equal to 0 and less than 1, there is always a 0 in front of the decimal point. For the remaining steps this constant is omitted to save space. There is always a 1 after the decimal point, except for 0 and intermediate results. This bit is encoded so the DSP loses one additional bit of precision compared with IEEE.

```
mant (500.0, DSP, cut) = .1111101000...
```

The exponent must also be converted to binary format:

```
exp (500.0, DSP) = 9 == 09 (hex) == 1001 (bin)
```

Negative exponents are encoded by the base 2 representation of the positive value.

The sign is encoded into the mantissa by taking the two's complement for negative numbers and placing a bit set to 1 in the front. For DSP and positive numbers, a bit cleared to 0 is added at the front.

```
mant(500.0, DSP) = 0111110100000000 (bin)
```

The two's complement is taken for negative numbers:

```
mant(-500.0, DSP) = 1000001100000000 (bin)
```

Finally the mantissa and the exponent must be joined according to the following listing.

Listing: Representation of Decimal 500.0 in DSP

```
  = 7D 00 (mantissa) 00 09 (exponent) (DSP as hex)
  = 7D 00 00 09 (DSP as hex)

  = 0111 1101 0000 0000 0000 0000 0000 1001 (DSP as binary)
```

The following listing shows the DSP representation of decimal -500.

Listing: Representation of Decimal -500.0 in DSP

```
  = 83 00 (mantissa) 00 09 (exponent) (DSP as hex)
  = 83 00 00 09 (DSP as hex)

  = 1000 0011 0000 0000 0000 0000 0000 1001 (DSP as binary)
```

NOTE

The order of the byte representation of a floating-point value depends on the byte ordering of the backend. Consider the first byte in the previous diagrams as the most significant byte.

10.4 Volatile Objects and Absolute Variables

The Compiler does not do register and constant tracing on volatile or absolute global objects. Accesses to volatile or a bsolute global objects are not eliminated. See the following listing for one reason to use a volatile declaration.

Listing: Using a Volatile Declaration to Avoid an Adverse Side Effect

```
volatile int x;
void main(void) {

  x = 0;

  ...

  if (x == 0) { // without volatile attribute, the

                // comparison may be optimized away!

    Error();    // Error() is called without compare!

  }

}
```

10.5 Bitfields

There is no standard way to allocate bitfields. Bitfield allocation varies from compiler to compiler, even for the same target. Using bitfields for access to I/O registers is non-portable and inefficient for the masking involved in unpacking individual fields. It is recommended that you use regular bit-and (&) or bit-or (|) operations for I/O port access.

The maximum width of bitfields depends on the backend in that plain `int` bitfields are signed (see HC(S)08 Backend for details). As stated in Kernighan and Ritchie's *The C Programming Language*, 2 **nd** ed., the use of bitfields is equivalent to using bit masks to which the operators &, |, ~, |=, or &= are applied. In fact, the Compiler translates bitfield operations to bit mask operations.

The topic covered here:

- Signed Bitfields

10.5.1 Signed Bitfields

A common mistake is to use signed bitfields, but testing them as if they were unsigned. Signed bitfields have a value -1 or 0. Consider the following example.

Listing: Testing a Signed Bitfield as Unsigned

```
typedef struct _B {
  signed int b0: 1;} B;

  B b;

if (b.b0 == 1) ...
```

The Compiler issues a warning and replaces the 1 with -1 because the condition `(b.b0 == 1)` does not make sense, that is, it is always false. The test `(b.b0 == -1)` performs as expected. This substitution is not ANSI compatible and will not be performed when the *-Ansi: Strict ANSI* compiler option is active.

Use an unsigned bitfield to test.Unsigned bitfields have the values 0 or 1, as shown in the following listing.

Listing: Using Unsigned Bitfields

```
typedef struct _B {
  unsigned b0: 1;

} B;

  B b;

  if (b.b0 == 1) ...
```

Because `b0` is an unsigned bitfield having a value 0 or 1, the test `(b.b0 == 1)` is correct.

NOTE

To save memory, it is recommended implementing globally accessible boolean flags as unsigned bitfields of width 1. However, it is not recommended using bitfields for other

purposes because using bitfields to describe a bit pattern in memory is not portable between compilers, even on the same target, as different compilers may allocate bitfields differently.

For information about bitfield allocation by the Compiler, see the Data Types section in HC(S)08 Backend .

10.6 Segmentation

The Linker supports the concept of segments in that you may partition the memory space into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions which the Linker then allocates into that segment. An entry in the Linker parameter file determines where that segment actually lies.

Listing: Syntax of the Segment Specification Pragma

```
SegDef= "#pragma" SegmentType ({SegmentMod} SegmentName |
                                            DEFAULT)

SegmentType= CODE_SEG|CODE_SECTION|

             DATA_SEG|DATA_SECTION|

             CONST_SEG|CONST_SECTION|

             STRING_SEG|STRING_SECTION

SegmentMod= __DIRECT_SEG|__NEAR_SEG|__CODE_SEG

     |__FAR_SEG|__BIT_SEG|__Y_BASED_SEG

     |__Z_BASED_SEG|__DPAGE_SEG|__PPAGE_SEG

     |__EPAGE_SEG|__RPAGE_SEG|__GPAGE_SEG"

     |__PIC_SEG|__LINEAR_SEG|CompatSegmentMod

CompatSegmentMod=DIRECT|NEAR|CODE|FAR|BIT|Y_BASED|Z_BASED|

                 DPAGE|PPAGE|EPAGE|RPAGE|GPAGE|PIC
```

The two basic types of segments, code and data segments, require two pragmas for segment specification:

- `#pragma CODE_SEG <segment_name>`
- `#pragma DATA_SEG <segment_name>`

In addition there are pragmas for constant data and for strings:

- `#pragma CONST_SEG <segment_name>`
- `#pragma STRING_SEG <segment_name>`

All four pragmas remain valid until the next pragma of the same kind is encountered.

In the HIWARE object file format, the Linker puts constants into DATA_SEG unless you specify a CONST_SEG. In the ELF Object file format, the Linker always puts constants into a constant segment.

The Linker puts strings into the STRINGS segment until you specify a STRING_SEG pragma. After this pragma, the Linker allocates all strings into this constant segment. The linker then treats this segment like any other constant segment.

If you do not specify a segment, the Compiler assumes two default segments named DEFAULT_ROM (the default code segment) and DEFAULT_RAM (the default data segment). Use the segment name DEFAULT to explicitly make these default segments the current segments:

```
#pragma CODE_SEG DEFAULT

#pragma DATA_SEG DEFAULT

#pragma CONST_SEG DEFAULT

#pragma STRING_SEG DEFAULT
```

You may also declare segments as __SHORT_SEG by inserting the keyword __SHORT_SEG just before the segment name. This makes the Compiler use short absolute addresses (8 bits or 16 bits, depending on the Backend) to access global objects or to call functions. It is the programmer's responsibility to allocate __SHORT_SEG segments in the proper memory area.

NOTE

You may not declare the DEFAULT code and data segments as __SHORT_SEG.

The backend specifies the meaning of the other segment modifiers, such as __NEAR_SEG and __FAR_SEG. The backend ignores unsupported modifiers. Refer to HC(S)08 Backend for information about modifier support.

The segment pragmas also affect static local variables. Static local variables are local variables with the *static* flag set. They are in fact normal global variables but with scope limited to the function in which they are defined:

```
#pragma DATA_SEG MySeg

static char myfun(void) {

  static char i = 0; /* place this variable into MySeg */
```

```
  return i++;


}


#pragma DATA_SEG DEFAULT
```

NOTE

Using the ELF/DWARF object file format (-F1 or -F2 compiler option), the Linker places all constants into the .rodata section by default unless you specify a `#pragma CONST_SEG`.

NOTE

Aliases that satisfy the ELF naming convention are available for all segment names. Use `CODE_SECTION` instead of `CODE_SEG`. Use `DATA_SECTION` instead of `DATA_SEG`. Use `CONST_SECTION` instead of `CONST_SEG`. Use `STRING_SECTION` instead of `STRING_SEG`. These aliases behave exactly as do the `XXX_SEG` name versions.

Listing: Example of Segmentation Without the -Cc Compiler Option

```
                                /* Placed into Segment: */
static int a;                   /* DEFAULT_RAM(-1) */

static const int c0 = 10;       /* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg

static int b;                   /* MyVarSeg(0) */

static const int c1 = 11;       /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT

static int c;                   /* DEFAULT_RAM(-1) */

static const int c2 = 12;       /* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg

#pragma CONST_SEG MyConstSeg

static int d;                   /* MyVarSeg(0)   */

static const int c3 = 13;       /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT

static int e;                   /* DEFAULT_RAM(-1) */

static const int c4 = 14;       /* MyConstSeg(1)   */

#pragma CONST_SEG DEFAULT

static int f;                   /* DEFAULT_RAM(-1) */

static const int c5 = 15;       /* DEFAULT_RAM(-1) */
```

Listing: Example of Segmentation with the -Cc Compiler Option

```
                                /* Placed into Segment: */
static int a;                   /* DEFAULT_RAM(-1) */

static const int c0 = 10;       /*
ROM_VAR(-2)     */

#pragma DATA_SEG MyVarSeg

static int b;                   /* MyVarSeg(0) */

static const int c1 = 11;       /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT

static int c;                   /* DEFAULT_RAM(-1) */

static const int c2 = 12;       /* ROM_VAR(-2)     */

#pragma DATA_SEG MyVarSeg

#pragma CONST_SEG MyConstSeg

static int d;                   /* MyVarSeg(0)   */

static const int c3 = 13;       /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT

static int e;                   /* DEFAULT_RAM(-1) */

static const int c4 = 14;       /* MyConstSeg(1)   */

#pragma CONST_SEG DEFAULT

static int f;                   /* DEFAULT_RAM(-1) */

static const int c5 = 15;       /* ROM_VAR(-2) */
```

10.7 Optimizations

The Compiler applies a variety of code-improving techniques under the term `optimization`. This section provides a short overview about the most important optimizations.

The topics covered here are as follows:

- Peephole Optimizer
- Strength Reduction
- Shift Optimizations
- Branch Optimizations
- Dead-Code Elimination
- Constant-Variable Optimization
- Tree Rewriting

10.7.1 Peephole Optimizer

The Compiler contains a simple optimizer, called a peephole optimizer. A peephole optimizer optimizes specific co de patterns for speed or code size. After recognizing these specific patterns, the Compiler replaces them with other optimized patterns.

After the Compiler backend generates the optimized code, the generated code may still benefit from further optimization. The peephole optimizer generates backend-dependent code, because optimizer implementation uses the characteristic code patterns of the specific backend.

Certain peephole optimizations only make sense in conjunction with other optimizations, or together with some code patterns. These patterns may have been generated by doing other optimizations. The peephole optimizer removes some optimizations (for example, removing a branch to the next instruction), although the branch optimizer can perform these optimizations as well. The peephole optimizer performs such simple branch optimizations to reach new optimizable states.

10.7.2 Strength Reduction

Strength reduction optimizes by replacing expensive operations with cheaper ones, where the cost factor is either execution time or code size. Examples include replacing multiplication and division by constant powers of two with left or right shifts.

NOTE

The Compiler can only replace a division by two with a shift operation when the target division is implemented such that -1/2 == -1, or the dividend is unsigned. Negative values produce different results. The Compiler can use a shift when the C source code already contains a shift, or the value to be shifted is unsigned.

10.7.3 Shift Optimizations

When shifting a byte variable by a constant number of bits, the Compiler attempts to implement such shifts in the most efficient way.

10.7.4 Branch Optimizations

This o ptimization minimizes the span of branch instructions. The Compiler generates a short branch rather than a long branch whenever possible. Also when possible, the Compiler resolves branches to branches into two branches to the same target. When possible the Compiler removes redundant branches (for example, a branch to the instruction immediately following it).

10.7.5 Dead-Code Elimination

The Compiler removes dead assignments while generating code. In some programs it may find additional cases of unused expressions.

10.7.6 Constant-Variable Optimization

When any expressio n uses a constant non-volatile variable, the Compiler replaces it by the constant value the variable holds. This requires less code than taking the object itself.

If no expression takes the address of the constant non-volatile object, the Compiler removes the object itself (notice `ci` in the following listing). This uses less memory space.

Listing: Example Demonstrating Constant Variable Optimization

```
void f(void) {
  const int ci  = 100; // ci removed (no address taken)

  const int ci2 = 200; // ci2 not removed (address taken below)

  const volatile int ci3 = 300; // ci3 not removed (volatile)

  int i;

  int *p;

  i = ci;   // replaced by i = 100;

  i = ci2;  // no replacement

  p = &ci2; // address taken

}
```

The Compiler does not remove global constant non-volatile variables. They are replaced in expressions by the constant value they hold.

The Compiler optimizes constant non-volatile arrays (notice `array[]` in the following listing).

Listing: Example Demonstrating Constant, Non-Volatile Array Optimization

```
void g(void) {
  const int array[] = {1,2,3,4};

  int i;

  i = array[2]; // replaced by i=3;

}
```

10.7.7 Tree Rewriting

The structure of the intermediate code between Frontend and Backend allows the Compiler to perform some optimizations on a higher level. Examples are shown in the following sections:

- Switch Statements
- Absolute Values
- Combined Assignments

10.7.7.1 Switch Statements

Any C Compiler requires efficient translation of switch statements. The Compiler applies different strategies, that is, branch trees, jump tables, and a mixed strategy, depending on the case label values and their numbers. The following table describes how the Compiler implements these strategies.

Table 10-6. Switch Implementations

Method	Description
Branch Sequence	For small switches with scattered case label values, the Compiler generates an if ... elsif ... elsif ... else ... sequence if the Compiler switch -Os is active.
Branch Tree	For small switches with scattered case label values, the Compiler generates a branch tree. This is equivalent to unrolling a binary search loop of a sorted jump table and

Table continues on the next page...

Table 10-6. Switch Implementations (continued)

Method	Description
	therefore is very fast. However, there is a point at which this method is not feasible simply because it uses too much memory.
Jump Table	When the branch tree method is not feasible, the Compiler creates a table plus a call of a switch processor. There are two different switch processors. The Compiler uses a direct jump table when there are a lot of labels with more or less consecutive values, and a binary search table when the label values are scattered.
Mixed Strategy	In some cases switches may have clusters of consecutive label values separated by other labels with scattered values. In this case, the Compiler applies a mixed strategy, generating branch trees or search tables for the scattered labels and direct jump tables for the clusters.

10.7.7.2 Absolute Values

Calculating absolute values c alls for optimization on a higher level. In C, the programmer writes something similar to:

```
float x, y;


x = (y < 0.0) ? -y : y;
```

This results in lengthy and inefficient code. The Compiler recognizes cases like this and treats them specially in order to generate the most efficient code. Only the most significant bit must be cleared.

10.7.7.3 Combined Assignments

The Compiler can also recogniz e the equivalence between the three following statements:

```
x = x + 1;
```

```
x += 1;
```

```
x++;
```

and between:

```
x =
x / y;
```

```
x /= y;
```

Therefore, the Compiler generates equally efficient code for either case.

10.8 Using Qualifiers for Pointers

This section provides some examples for the use of const or volatile, commonly used qualifiers in Embedded Programming.

Consider the following example:

```
int i;
```

```
const int ci;
```

The above definitions are: a *normal* variable `i` and a *constant* variable `ci`. The Compiler places each into ROM. Note that for C++, you must initialize the constant `ci`.

```
int *ip;
```

```
const int *cip;
```

In this case, `ip` is a pointer to an `int`, and `cip` is a pointer to a `const int`.

```
int *const icp;


const int *const cicp;
```

Here, `icp` is a const pointer to an `int`, where `cicp` is a const pointer to a `const int`.

The qualifier for such pointers always appears on the right side of the *.

You can express this rule in the same way for the volatile qualifier. Consider the following example of an array of five constant pointers to volatile integers:

```
volatile int *const arr[5];
```

The array `arr` contains five constant pointers pointing to volatile integers. Because the array itself is constant, it is put into ROM. Whether the array is constant or not does not change where the pointers point. Consider the next example:

```
const char *const *buf[] = {&a, &b};
```

Initializing `buf` makes it a non-constant array. This array contains two pointers which point to constant characters. Because the array is non-constant, neither the Compiler nor the Linker can place `buf` into ROM.

Consider a constant array of five ordinary function pointers:

```
void (*fp)(void);
```

This shows a function pointer `fp` returning void and having `void` as parameter. Define the pointer with:

```
void (*fparr[5])(void);
```

You can also use a `typedef` to separate the function pointer type and the array:

```
typedef void (*Func)(void);
```

```
Func fp;


Func fparr[5];
```

You can write a constant function pointer as:

```
void (*const cfp) (void);
```

Consider a constant function pointer having a constant `int` pointer as a parameter returning `void`:

```
void (*const cfp2) (int *const);
```

Or a `const` function pointer returning a pointer to a `volatile double` having two constant integers as parameter:

```
volatile double *(*const fp3) (const int, const int);
```

And one more:

```
void (*const fp[3])(void);
```

This is an array of three constant function pointers, having `void` as parameter and returning void. The Compiler allocates `fp` in ROM because the `fp` array is constant.

Consider an example using function pointers:

```
int (* (** func0(int (*f) (void))) (int (*) (void))) (int (*)
(void)) {


  return 0;


}
```

This function, called `func()`, has one function pointer argument called `f`. The return value is a complex function pointer. Here, we do not explain where to put a `const`, so that the destination of the returned pointer cannot be modified. Alternately, write the same function more simply using `typedef`s (refer the following listing).

Listing: Using typedefs

```
typedef int (*funcType1) (void);
typedef int (* funcType2) (funcType1);

typedef funcType2 (* funcType3) (funcType1);

funcType3* func0(funcType1 f) {

  return 0;

}
```

In this case the places of the `const` become obvious. Just behind the * in `funcType3`:

```
  typedef funcType2 (* const constfuncType3) (funcType1);


  constfuncType3* func1(funcType1 f) {


    return 0;


  }
```

In the first version, place the `const` here:

```
  int (* (*const * func1(int (*f) (void))) (int (*) (void)))

                                          (int (*) (void)) {


    return 0;


  }
```

10.9 Defining C Macros Containing HLI Assembler Code

You can define some ANSI C macros that contain HLI assembler statements when working with the HLI assembler. Because the HLI assembler is heavily Backend-dependent, the following example uses a pseudo Assembler Language.

Listing: Coding Example

```
CLR Reg0       ; Clear Register zero
CLR Reg1       ; Clear Register one

CLR var        ; Clear variable `var' in memory

LOAD var,Reg0  ; Load the variable `var' into Register 0

LOAD #0, Reg0  ; Load immediate value zero into Register 0

LOAD @var,Reg1 ; Load address of variable `var' into Reg1

STORE Reg0,var ; Store Register 0 into variable `var'
```

The HLI instructions above are only a possible solution. For real applications, replace the pseudo HLI instructions above with the HLI instructions for your target.

- Defining Macro
- Using Macro Parameters
- Using Immediate-Addressing Mode in HLI Assembler Macros
- Generating Unique Labels in HLI Assembler Macros
- Generating Assembler Include Files (-La Compiler Option)

10.9.1 Defining Macro

Use the `define` preprocessor directive to define an HLI assembler macro.

The following listing defines a macro that clears the R0 register.

Listing: Defining the ClearR0 Macro

```
/* The following macro clears R0. */
#define ClearR0 {__asm CLR R0;}
```

The source code invokes the ClearR0 macro in the following manner:

```
ClearR0;
```

Then the preprocessor expands the ClearR0 macro:

```
{ __asm CLR R0 ; } ;
```

An HLI assembler macro can contain one or several HLI assembler instructions. As the ANSI-C preprocessor expands a macro on a single line, you cannot define an HLI assembler block in a macro. You can, however, define a list of HLI assembler instructions (refer the following listing).

Listing: Defining Two Macros on the Same Line of Source Code

```
/* The following macro clears R0 and R1. */
#define ClearR0and1 {__asm CLR R0; __asm CLR R1; }
```

The source code invokes this macro in the following way:

```
ClearR0and1;
```

The preprocessor expands the macro:

```
{ __asm CLR R0 ; __asm CLR R1 ; } ;
```

You can define an HLI assembler macro on several lines using the line separator \.

NOTE

This may enhance the readability of your source file. However, the ANSI-C preprocessor still expands the macro on a single line.

Listing: Defining a Macro on More than One Line of Source Code

```
/* The following macro clears R0 and R1. */
#define ClearR0andR1 {__asm CLR R0; \

                      __asm CLR R1;}
```

The source code invokes the macro in the following way:

```
ClearR0andR1;
```

The preprocessor expands the ClearR0andR1 macro:

```
{__asm CLR R0; __asm CLR R1; };
```

10.9.2 Using Macro Parameters

An HLI assembler macro may have some parameters which are referenced in the macro code. The following listing defines the Clear1 macro using the `var` parameter.

Listing: Clear1 Macro Definition

```
/* This macro initializes the specified variable to 0.*/
#define Clear1(var) {__asm CLR var;}
```

Invoking the Clear1 macro in the source code:

```
Clear1(var1);
```

The preprocessor expands the Clear1 macro:

```
{__asm CLR var1 ; };
```

10.9.3 Using Immediate-Addressing Mode in HLI Assembler Macros

An ambiguity exists when using the immediate addressing mode within a macro.

For the ANSI-C preprocessor, the symbol # inside a macro specifically indicates a string constructor. Using #pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing instructs the Compiler that in all subsequent macros, the instructions remain unchanged whenever the symbol # is specified. This macro is valid for the rest of the file in which it is specified.

Listing: Definition of the Clear2 Macro

```
/* This macro initializes the specified variable to 0.*/
#pragma NO_STRING_CONSTR

#define Clear2(var){__asm LOAD #0,Reg0;__asm STORE Reg0,var;}
```

Invoking the `Clear2` macro in the source code:

```
Clear2(var1);
```

The preprocessor expands the Clear2 macro:

```
{ __asm LOAD #0,Reg0;__asm STORE Reg0,var1; };
```

10.9.4 Generating Unique Labels in HLI Assembler Macros

Invoking the same m acro twice in the same function causes the ANSI C preprocessor to generate the same label twice (once in each macro expansion). Use the special string concatenation operator of the ANSI-C preprocessor (`##`) to generate unique labels. Refer the following listing.

Listing: Using the ANSI C Preprocessor String Concatenation Operator

```
/* The following macro copies the string pointed to by 'src'
   into the string pointed to by 'dest'.

   'src' and 'dest' must be valid arrays of characters.

   'inst' is the instance number of the macro call. This

   parameter must be different for each invocation of the

   macro to allow the generation of unique labels. */

#pragma NO_STRING_CONSTR

#define copyMacro2(src, dest, inst) { \

  __asm           LOAD   @src,Reg0;  /* load src addr */ \

  __asm           LOAD   @dest,Reg1; /* load dst addr */ \

  __asm           CLR    Reg2;        /* clear index reg */ \

  __asm lp##inst: LOADB  (Reg2, Reg0); /* load byte reg indir */  \

  __asm           STOREB (Reg2, Reg1); /* store byte reg indir */ \

  __asm           ADD    #1,Reg2; /* increment index register */   \

  __asm           TST    Reg2;     /* test if not zero */           \

  __asm           BNE    lp##inst; }
```

Invoking the `copyMacro2` macro in the source code:

```
  copyMacro2(source2, destination2, 1);


  copyMacro2(source2, destination3, 2);
```

During expansion of the first macro, the preprocessor generates an `lp1` label. During expansion of the second macro the preprocessor creates an `lp2` label.

10.9.5 Generating Assembler Include Files (-La Compiler Option)

In many projects it often makes sense to use both a C compiler and an assembler. Both have different advantages. The compiler uses portable and readable code while the assembler provides full control for time-critical applications, or for directly accessing the hardware.

However, the compiler cannot read `include` files of the assembler, and the assembler cannot read the `header` files of the compiler.

The Compiler produces an assembler include file that allows both tools to use one single source to share constants, variables, labels, and even structure fields.

The compiler writes an output file in assembler format which contains all information needed by a C header file.

The current implementation supports the following mappings:

- Macros

  C defines translate to assembler `EQU` directives.

- `enum` values

  C `enum` values translate to `EQU` directives.

- C types

  Generates the size of any type and the offset of structure fields for all `typedef`s. For bitfield structure fields, generates the bit offset and the bit size

- Functions

  Generates an XREF entry for each function.

- Variables

  Generates C variables with an `XREF`. In addition, defines all fields with an `EQU` directive for structures or unions.

- Comments

  Includes C style comments (`/* ... */`) as assembler comments (`;....`).

The topics covered here are as follows:

- General
- Macros

- Enumerations
- Types
- Functions
- Variables
- Comments
- Guidelines

10.9.5.1 General

You must specially prepare a header file to generate the assembler include file. A pragma anywhere in the header file can enable assembler output:

```
#pragma CREATE_ASM_LISTING ON
```

The Compiler generates only those macro definitions and declarations subsequent to this pragma. The compiler stops generating elements when #pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing occurs with an OFF parameter.

```
#pragma CREATE_ASM_LISTING OFF
```

Not all entries generate legal assembler constructs. The compiler does not check for legal assembler syntax when translating macros. Put macros containing elements not supported by the assembler in a section controlled by `#pragma CREATE_ASM_LISTING OFF`.

The compiler only creates an output file when the `-La` option is specified and the compiled sources contain `#pragma CREATE_ASM_LISTING ON` (refer the following listing).

Listing: Header File: a.h

```
#pragma CREATE_ASM_LISTING ON
typedef struct {

  short i;

  short j;

} Struct;

Struct Var;

void f(void);

#pragma CREATE_ASM_LISTING OFF
```

When the compiler reads this header file with the `-La=a.inc a.h` option, it generates the following.

Listing: a.inc File

```
Struct_SIZE     EQU $4
Struct_i        EQU $0

Struct_j        EQU $2

                XREF Var

Var_i           EQU Var + $0

Var_j           EQU Var + $2

                XREF f
```

You can now use the assembler INCLUDE directive to include this file into any assembler file. The content of the C variable, Var_i, can also be accessed from the assembler without any uncertain assumptions about the alignment used by the compiler. Also, whenever a field is added to the structure Struct, the assembler code must not be altered. You must, however, regenerate the a.inc file with a make tool.

The Compiler does not create the assembler include file every time it reads the header file, but only when the header file changes significantly. Specify the -La option only when the compiler must generate a.inc. If -La is always present, a.inc is always generated. A make tool always restarts the assembler because the assembler files depend on a.inc. Such a makefile might be similar to the following listing.

Listing: Sample Makefile

```
a.inc : a.h
  $(CC) -La=a.inc a.h

a_c.o : a_c.c a.h

  $(CC) a_c.c

a_asm.o : a_asm.asm a.inc

  $(ASM) a_asm.asm
```

The order of elements in the header file is the same as the order of the elements in the created file, except that comments may be inside elements in the C file. In this case, the comments may be before or after the whole element.

The order of defines does not matter for the compiler. The order of EQU directives does matter for the assembler. If the assembler has problems with the order of EQU directives in a generated file, you must change the corresponding header file accordingly.

10.9.5.2 Macros

The Compiler translates defines lexically and not semantically. The compiler does not check the accuracy of the define. The following example shows some uses of this feature:

Listing: Example Source Code

```
#pragma CREATE_ASM_LISTING ON
int i;

#define UseI i

#define Constant 1

#define Sum Constant+0X1000+01234
```

The source code in the above listing produces the following output:

Listing: Output

```
                        XREF  i
UseI                    EQU   i

Constant                EQU   1

Sum                        EQU   Constant + $1000 + @234
```

Disassembly translates the hexadecimal C constant `0x1000` to `$1000` and translates the octal `01234` to `@1234`. In addition, the compiler inserts one space between every two tokens. The Compiler makes no other changes in the assembler listing for defines.

Some macros compile but do not assemble. The Compiler compiles macros with parameters, predefined macros, and macros with no defined value, but the assembler produces no output for these macros. Although these macros are in the header file the Compiler uses to generate the assembler include file, locate the macros together and preface the section with `#pragma CREATE_ASM_LISTING OFF`, to prevent the compiler from generating the assembly listing.

The following defines do not work or are not generated:

Listing: Improper Defines

```
#pragma CREATE_ASM_LISTING ON
int i;

#define AddressOfI &i

#define ConstantInt ((int)1)

#define Mul7(a) a*7

#define Nothing

#define useUndef UndefFkt*6

#define Anything § § / % & % / &  + * % ç 65467568756 86
```

The source code in the above listing produces the following output.

Listing: Output

```
                        XREF  i
AddressOfI              EQU   & i

ConstantInt             EQU   ( ( int ) 1 )

useUndef                EQU   UndefFkt * 6

Anything                   EQU   § § / % & % / & + * % ç 65467568756 86
```

The assembler does not assemble the `AddressOfI` macro because the assembler cannot interpret the `&` C address operator. Also, do not use other C-specific operators such as dereferenciation (`*ptr`). The compiler generates them into the assembler listing file without any translation.

The `ConstantInt` macro does not assemble because the assembler does not know the cast syntax and the types. The assembler does not write macros with parameters (such as `Mu17`) or macros with no actual value to the listing.

The C preprocessor ignores the syntactical content of the macro, therefore the compiler processes macros like `useUndef`, with the undefined object `UndefFkt`, correctly. The assembler `EQU` directive requires definitions for all used objects.

The Compiler processes the `Anything` macro in the previous listing with no difficulty. The assembler cannot process these random characters.

10.9.5.3 Enumerations

An `enum` in C has a unique name and a defined value (refer the following listing).

Listing: Enumerations

```
#pragma CREATE_ASM_LISTING ON
enum {

 E1=4,

 E2=47,

 E3=-1*7

};
```

The Compiler generates `enums` as `EQU` directives (refer the following listing).

Listing: Enumerations

```
E1                      EQU $4
E2                      EQU $2F

E3                      EQU $FFFFFFF9
```

NOTE

The Compiler generates negative values as 32-bit hex numbers.

10.9.5.4 Types

There is no reason to generate the size of every occurring type, therefore only `typedef`s are considered here.

The size of the newly defined type is specified for all `typedef`s.

Listing: typedefs

```
#pragma CREATE_ASM_LISTING ON
typedef long LONG;

struct tagA {

  char a;

  short b;

};

typedef struct {

  long d;

  struct tagA e;

  int f:2;

  int g:1;

} str;
```

The Compiler appends the term `_SIZE` to the end of the `typedef`'s name for the size name. For structures, the Compiler generates the offset of all structure fields relative to the structure's start, and generates the structure offset names by appending an underscore (_) and the structure field's name to the name of the `typedef` as shown in the following listing.

Listing:Disassembly of above listing

```
LONG_SIZE               EQU $4
str_SIZE                EQU $8

str_d                   EQU $0

str_e                   EQU $4

str_e_a                 EQU $4

str_e_b                 EQU $5

str_f                   EQU $7

str_f_BIT_WIDTH         EQU $2
```

```
str_f_BIT_OFFSET        EQU $0

str_g                   EQU $7

str_g_BIT_WIDTH         EQU $1

str_g_BIT_OFFSET        EQU $2
```

The Compiler contains all structure fields within that structure. The generated name contains all the names for all fields listed in the path. The Compiler ignore any element without a name (for example, an anonymous union).

The Compiler also generates the width and offset for all bitfield members. The offset 0 specifies the least significant bit, which is accessed with mask `0x1`. The offset 2 specifies the most significant bit, which is accessed with mask `0x4`. The width specifies the number of bits.

The offsets, bit widths, and bit offsets given here are examples. Different compilers may generate different values. In C, the Compiler determines the structure alignment and bitfield allocation and specifies the correct values.

10.9.5.5 Functions

The `XREF` directive generates declared functions, enabling them to be used with the Assembler. Do not generate the function into the output file, but place the function in an area of your code preceded by `#pragma CREATE_ASM_LISTING OFF`. The Assembler does not allow the redefinition of labels declared with `XREF` (see the following listing).

Listing: Function Prototyping with the CREATE_ASM_LISTING Pragma

```
#pragma CREATE_ASM_LISTING ON
void main(void);

void f_C(int i, long l);

#pragma CREATE_ASM_LISTING OFF

void f_asm(void);
```

The source code above disassembles as in the following listing.

Listing: Disassembly of the following

```
  XREF main
  XREF f_C
```

10.9.5.6 Variables

The following listing shows variable examples.

Listing: Examples of Variables

```
#pragma CREATE_ASM_LISTING ON
struct A {

  char a;

  int i:2;

};

struct A VarA;

#pragma DATA_SEG __SHORT_SEG ShortSeg

int VarInt;
```

Declare variables with XREF. In addition, for structures, EQU directives define every field. For bitfields, define the bit offset and bit size.

Define variables in the __SHORT_SEG segment with XREF.B, Which informs the assembler about the direct access. EQU.B directives define the structure fields in __SHORT_SEG segments.

Listing: Examples of Variables

```
                        XREF VarA
VarA_a                  EQU VarA + $0

VarA_i                  EQU VarA + $1

VarA_i_BIT_WIDTH        EQU $2

VarA_i_BIT_OFFSET       EQU $0

                        XREF.B VarInt
```

The Compiler does not write the variable size explicitly. To generate the variable size, use a typedef with the variable type.

The offsets, bit widths, and bit offsets given here are examples. Different compilers may generate different values. In C, the Compiler determines the structure alignment and the bitfield allocation and specifies the correct values.

10.9.5.7 Comments

When creating an assembly l isting, the Compiler writes any comments in the assembler include file. When the Compiler encounters comments within a typedef, structure, or variable declaration, the Compiler places the comments either before or after the declaration in the generated file. The Compiler never places comments inside the declaration, even when the declaration contains multiple lines. Therefore, a comment after a structure field within a typedef appears before or after the whole typedef, not just

after the structure field. Every comment is on a single line. An empty comment (`/* */`) inserts an empty line into the created file. Refer the following listing for an example of using constants.

Listing: Example Using Comments

```
#pragma CREATE_ASM_LISTING ON
/*

 The function main is called by the startup code.

 The function is written in C. Its purpose is

 to initialize the application. */

void main(void);

/*

  The SIZEOF_INT macro specified the size of an integer type

  in the compiler. */

typedef int SIZEOF_INT;

#pragma CREATE_ASM_LISTING OFF
```

The following listing shows the listing above when disassembled.

Listing: Example Using Comments

```
; The function main is called by the startup code.
; The function is written in C. Its purpose is

; to initialize the application.

                     XREF main

;

;   The SIZEOF_INT macro specified the size of an integer type

;   in the compiler.

SIZEOF_INT_SIZE         EQU $2
```

10.9.5.8 Guidelines

The `-La` option translates specified parts of header files into an `include` file to import labels and defines into an assembler source. Because the -La compiler option is very powerful, use the following guidelines to avoid incorrect usage. This section describes using this option to combine C and assembler sources, both using common header files.

The following general implementation recommendations help to avoid problems when writing software using the common header file technique.

- Make all interface memory reservations or definitions in C source files. Memory areas only accessed from assembler files can still be defined in the common assembler manner.
- Compile only C header files (and not the C source files) with the -La option to avoid multiple defines and other problems. The project-related makefile must contain an inference-rules section that defines the C header file-dependent include files to be created.
- Use `#pragma CREATE_ASM_LISTING ON/OFF` only in C header files. This pragma selects the objects to translated to the assembler `include` file. The created file then holds the corresponding assembler directives.
- Do not use the `-La` option as part of the command line options used for all compilations. Use this option in combination with the `-Cx` (no Code Generation) option. Without this option, the compiler creates an object file which could accidently overwrite a C source object file.
- Remember to extend the list of dependencies for assembler sources in your make file.
- Ensure that the compiler-created assembler `include` file is included into your assembler source.

NOTE

The Compiler translates zero-page declared objects (if this is supported by the target) into `XREF.B` directives for the base address of a variable or constant. The compiler translates structure fields in the zero page into an EQU.B directive to access them. Explicit zero-page addressing syntax may be necessary as some assemblers use extended addresses to EQU.B defined labels.

Chapter 11
Generating Compact Code

The Compiler generates compact and efficient code whenever possible, although not everything is handled directly by the Compiler. Denser code is possible by specifying certain Compiler options, or `__SHORT_SEG` segments (if available).

This chapter covers the following topics:

11.1 Compiler Options

Using the following compiler options helps reduce the size of the code generated. Note that some options may not be available for every target.

The topics covered here are as follows:

11.1.1 -Or: Register Optimization

When accessing pointer fields, this option prevents the compiler from reloading the pointer address for each access. An index register holds the pointer value across statements when possible.

NOTE

This option may not be available for all targets.

11.1.2 -Oi: Inline Functions

Use the inline keyword or the command line option `-Oi` for C/C++ functions (see -Oi: Inlining for more information). Defining a function before using it helps the Compiler to inline it:

```
/* OK */                        /* better! */

void myfun(void);                 void myfun(void) {

void main(void) {                 // ...

   myfun();                       }

}                               void main(void) {

void myfun(void) {                    myfun();

  // ...                        }

}
```

This also allows the compiler to use a relative branch instruction instead of an absolute branch instruction.

11.2 __SHORT_SEG Segments

Tools access variables on the direct page (between 0 and 0xFF) using direct addressing. The Compiler allocates some variables on the direct page if they are defined in a __SHORT_SEG segment (refer the following listing).

Listing: Allocate Frequently-Used Variables on the Direct Page

```
#pragma
DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;

#pragma DATA_SEG DEFAULT

unsigned int myvar3, myVar4
```

In the previous example, myVar1 and myVar2 are both accessed using direct addressing mode. Variables myVar3 and myVar4 are accessed using extended addressing mode.

When you define some exported variables in a __SHORT_SEG segment, you must also specify in the external declaration for these variables that they are allocated in a __SHORT_SEG segment. The External definition of the variable defined above looks like:

```
#pragma DATA_SEG __SHORT_SEG myShortSegment


extern unsigned int myVar1, myVar2;


#pragma DATA_SEG DEFAULT


extern unsigned int myvar3, myVar4
```

Place the segment on the direct page using the parameter (PRM) file (refer the following listing).

Listing: Linker Parameter File

```
LINK test.abs
NAMES test.o startup.o ansi.lib END

SECTIONS

    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;

    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;

    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
```

```
PLACEMENT

    DEFAULT_ROM                     INTO  MY_ROM;

    DEFAULT_RAM                     INTO  MY_RAM;

    _ZEROPAGE, myShortSegment       INTO  Z_RAM;

END

STACKSIZE 0x60

VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE

The Linker is case-sensitive. The segment name must be identical in the C and PRM files.

11.3 Defining I/O Registers

The I/O Registers are usually based at address 0. To tell the compiler it must use direct addressing mode to access th e I/O registers, define these registers in a `__SHORT_SEG` section (if available) based at the specified address.

Define the I/O register in the C source file as in the following listing.

Listing: Definition of an I/O Register

```
typedef struct {
  unsigned char SCC1;

  unsigned char SCC2;

  unsigned char SCC3;

  unsigned char SCS1;

  unsigned char SCS2;

  unsigned char SCD;

  unsigned char SCBR;

} SCIStruct;

#pragma DATA_SEG __SHORT_SEG SCIRegs

SCIStruct   SCI;

#pragma DATA_SEG DEFAULT
```

Place the segment at the appropriate address in the PRM file (refer the following listing).

Listing: Linker Parameter File Allocating the I/O Register

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
```

```
SECTIONS
    SCI_RG = READ_WRITE  0x0013 TO 0x0019;
    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM                  INTO  MY_ROM;
    DEFAULT_RAM                  INTO  MY_RAM;
    _ZEROPAGE                    INTO  Z_RAM;
    SCIRegs                      INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE

The Linker is case-sensitive. The segment name must be identical in the C/C++ and PRM files.

11.4 Programming Guidelines

Following a few programming guidelines helps to reduce code size. The Compiler many things. However, a complex programming style or a style that forces the Compiler to perform special code sequences results in a less efficient optimization.

The topics covered here are as follows:

- Constant Function at Specific Address
- HLI Assembly
- Post- and Pre-Operators in Complex Expressions
- Boolean Types
- printf() and scanf()
- Bitfields
- Struct Returns
- Local Variables
- Parameter Passing
- Unsigned Data Types
- Inlining and Macros
- Data Types

- Short Segments
- Qualifiers

11.4.1 Constant Function at Specific Address

Sometimes functions are placed at a specific address, but the information regarding the functions is not available. The programmer knows the function starts at address 0x1234 and wants to call it. Without having the function definition, a linker error occurs because of the missing target function code. Use a constant function pointer to solve this problem:

```
void (*const fktPtr)(void) = (void(*)(void))0x1234;
void main(void) {
 fktPtr();
}
```

This produces efficient code with no linker errors. However, the function at `0x1234` must really exist.

The following code shows a better solution, without the need for a function pointer:

```
#define erase ((void(*)(void))(0xfc06))
void main(void) {
  erase(); /* call function at address 0xfc06 */
}
```

11.4.2 HLI Assembly

Do not mix High-Level Inline (HLI) Assembly with C declarations and statements (see the following listing). Using HLI assembly may affect the register trace of the compiler. The Compiler cannot touch HLI Assembly, and thus it is out of range for any optimizations except branch optimization.

Listing: Mixing HLI Assembly with C Statements (Not Recommended)

```
void myfun(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  __asm {
```

```
    /* some HLI statements */
  }
  /* maybe other C/C++ statements */
}
```

When encountering this code, the Compiler assumes that everything has changed. It cannot hold variables used by C statements in registers while processing HLI statements. Normally it is better to place special HLI code sequences into separate functions, although additional calls or returns may occur. Placing HLI instructions into separate functions (and modules) simplifies porting the software to another target (refer the following listing).

Listing: HLI Statements are Not Mixed with C Statements (Recommended)

```
/* hardware.c */
void special_hli(void) {
  __asm {
    /* some HLI statements */
  }
}
/* myfun.c */
void myfun(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  special_hli();
  /* maybe other C/C++ statements */
}
```

11.4.3 Post- and Pre-Operators in Complex Expressions

Writing a complex program results in complex code. In general it is the job of the compiler to optimize complex functions. Some rules may help the compiler to generate efficient code.

If the target does not support pre- or post-increment or pre- or post-decrement instructions, using the ++ and -- operators in complex expressions is not recommended. Post-increment and post-decrement particularly may result in additional code:

```
a[i++] = b[--j];
```

Write the above statement as:

```
j--; a[i] = b[j]; i++;
```

Use it in simple expressions as:

```
i++;
```

Avoid assignments in parameter passing or side effects (as ++ and --). The evaluation order of parameters is undefined (ANSI-C standard 6.3.2.2) and may vary from Compiler to Compiler, and even from one release to another:

```
i = 3;


myfun(i++, --i);
```

In the above example, `myfun()` is called either with myfun `(3,3)` or with `myfun(2,2)`.

11.4.4 Boolean Types

In C, the boolean type of an expression is an `int`. A variable or expression evaluated as 0 (zero) is FALSE and everything else (!= 0) is TRUE. Instead of using `int` (usually 16 or 32 bits), use an 8-bit type to hold a boolean result. For ANSI-C compliance, declare the basic boolean types in `stdtypes.h`:

```
typedef int Bool;


#define TRUE   1


#define FALSE 0
```

Use the following code to reduce memory usage and improve code density:

```
typedef Byte Bool_8;
```

from `stdtypes.h` (`Byte` is an unsigned 8-bit data type also declared in `stdtypes.h`).

11.4.5 printf() and scanf()

The `printf()` or `scanf()` code in the ANSI library can be reduced if no floating point support (`%f`) is used. Refer to the ANSI library reference and `printf.c` or `scanf.c` in your library for details about saving code (avoiding `float` or `doubles` in `printf()` may decrease code size by 50%).

11.4.6 Bitfields

Using bitfields to save memory generally produces additional code. For ANSI-C compliance, bitfields have a type of `signed int`, thus a bitfield of size 1 is either -1 or 0. This may force the compiler to sign extend operations:

```
struct {

  int b:0;  /* -1 or 0 */

} B;

int i = B.b; /* load the bit, sign extend it to -1 or 0 */
```

Sign extensions are normally time- and code-inefficient operations.

11.4.7 Struct Returns

When returning `structs`, the Compiler allocates space on the stack for the return value and then calls the function. Then the Compiler copies the return value to the variable `s`. During the return sequence, the Compiler copies the return value to `myfun` (refer to the listing displayed below).

Depending on the size of the `struct`, this may be done inline. After return, the caller `main` copies the result back into `s`. Depending on the Compiler or Target, it is possible to optimize some sequences (avoiding some copy operations). However, returning a `struct` by value may increase execution time, possibly increasing code and stack usage.

Listing: Returning a struct Forces the Compiler to Produce Lengthy Code

```
struct S
myfun(void)
  /* ... */

  return s; // (4)

}

void main(void) {

  struct S s;

  /* ... */

  s =
myfun();  // (1), (2), (3)

  /* ... */

}
```

With the following example, the Compiler passes the destination address and calls `myfun`. The callee, `myfun`, copies the result indirectly into the destination. This approach reduces stack usage, avoids copying `structs`, and results in denser code. Note that the Compiler may also inline the above sequence (if supported). But for rare cases the above sequence may not be exactly the same as returning the struct by value (for example, if `myfun` modifies the destination `struct`).

Listing: Pass a Pointer to the Callee for the Return Value

```
void
myfun(struct S *sp) {
  /* ... */

  *sp = s; // (4)

} void main(void) {

  S s;

  /* ... */


myfun(&s);  // (2)

  /* ... */
```

```
}
```

11.4.8 Local Variables

Using local variables instead of global variable results in better application manageability by reducing or avoiding side effects entirely. Using local variables or parameters reduces global memory usage but increases stack usage.

Target stack access capability influences code quality. Depending on the target capabilities, access to local variables may be very inefficient. Targets without a dedicated stack pointer require the use of an address register instead, making the address register unavailable for other values. Limited offsets or addressing modes causes inefficient variable access as well.

Allocating a large number of local variables causes the Compiler to generate a complex sequence to allocate the stack frame in the beginning of the function and to deallocate it at the end (refer the following listing).

Listing: Function with Many Local Variables

```
void myfun(void) {
  /* huge amount of local variables: allocate space! */

  /* ... */

  /* deallocate huge amount of local variables */

}
```

If the target provides special entry or exit instructions for such cases, allocation of many local variables is not a problem. You may also use global variables or static local variables. However, this deteriorates maintainability and may waste global address space.

The Compiler may offer an option to overlap parameter or local variables using a technique called *overlapping*, allocating local variables or parameters as globals. The linker overlaps them depending on their use. For targets with limited stack (for example, no stack addressing capabilities), this often is the only solution. However this solution makes the code non-reentrant (recursion is not allowed).

11.4.9 Parameter Passing

Avoid using parameters larger than the data registers size (see HC(S)08 Backend).

11.4.10 Unsigned Data Types

Use unsigned data types when possible as signed operations are more complex than unsigned operations (for example, shifts, divisions and bitfield operations). Take proper precautions to handle the value in the event it is less than zero.

11.4.11 Inlining and Macros

The inlining and macros are as follows:

- abs() and labs()
- memcpy() and memcpy2()

11.4.11.1 abs() and labs()

Use the corresponding macro `M_ABS` defined in `stdlib.h` instead of calling `abs()` and `absl()` in the `stdlib`:

```
/* extract
/* macro definitions of abs() and labs() */
#define M_ABS(j)  (((j) >= 0) ? (j) : -(j))
extern int     abs(int j);
extern long int labs(long int j);
```

But be careful as `M_ABS()` is a macro,

```
i = M_ABS(j++);
```

and is not the same as:

```
i = abs(j++);
```

11.4.11.2 memcpy() and memcpy2()

ANSI-C requires that the `memcpy()` library function in `strings.h` return a destination pointer and handle and can handle a count of zero:

Listing: Excerpts from the string.h and string.c Files Relating to memcpy()

```
/* extract of string.h *
extern void * memcpy(void *dest, const void * source, size_t count);

extern void  memcpy2(void *dest, const void * source, size_t count);

/* this function does not return dest and assumes count > 0 */

/* extract of string.c */

void * memcpy(void *dest, const void *source, size_t count) {

  uchar *sd = dest;

  uchar *ss = source;

  while (count--)

    *sd++ = *ss++;

  return (dest);

}
```

For a simpler, faster choice, use `memcpy2()` when the function does not have to return the destination or handle a count of zero (refer the following listing).

Listing: Excerpts from the string.c File Relating to memcpy2()

```
/* extract of string.c */
void

memcpy2(void *dest, const void* source, size_t count) {

  /* this func does not return dest and assumes count > 0 */

  do {

    *((uchar *)dest)++ = *((uchar*)source)++;

  } while(count--);

}
```

Replacing calls to `memcpy()` with calls to `memcpy2()` saves runtime and code size.

11.4.12 Data Types

Do not use larger data types than necessary. Use IEEE32 floating point format both for `float` and `doubles` if possible. Set the `enum` type to a smaller type than `int` using the `-T` option. Avoid data types larger than registers.

11.4.13 Short Segments

Whenever possible and available (not all targets support it), place frequently used global variables into a `DIRECT` or `__SHORT_SEG` segment using:

```
#pragma DATA_SEG __SHORT_SEG MySeg
```

11.4.14 Qualifiers

Use the `const` qualifier to help the compiler. In the HIWARE object-file format, the Compiler places `const` objects into ROM when the `-Cc` compiler option is given.

Chapter 12
HC(S)08 Backend

The Compiler's target-dependent Backend contains the code generator. This chapter discusses the technical details of the Backend for the HC(S)08 family.

12.1 Memory Models

A *memory model* is a code design strategy that organizes a program's code and data so that it best follows a particular addressing scheme used by the MCU.

NOTE

Further details on MCU memory models can be found in Banked Memory Support.

This section describes the following memory models:

- Banked Model
- SMALL Model
- TINY Model

12.1.1 Banked Model

The banked memory model uses the MCU's Memory Management Unit (MMU), allowing the extension of program space beyond the 64-kilobyte CPU-addressable memory map. Enabling `-MMU` and `-Mb` enables code banking and extends program space; enabling `-MMU` enables linear data access and extends data space (see -M (-Mb, -Ms, -Mt): Memory Model).

The topics covered are as follows:

12.1.1.1 Program Space Extension

Code banking extends program space using a paged memory system. The 8-bit `PPAGE` register selects the memory page, and a 16-kilobyte memory window to access the pages. Current architecture supports up to 256 pages of 16 kilobytes each, allowing up to 4 Megabytes of memory.

Some HC(S)08 derivatives support a memory expansion scheme. Using the paging mechanism, the user can access program memory by putting the page number into the `PPAGE` register, and computing the address offset from the starting address of the paging window. The following figure refers to MC9S08QE128 derivative as an example of memory access using the code banking model. Access addresses directly, using the CPU addresses, or use paged access, which uses the extended addressing method (for example, access `$06ABC` directly at `$06ABC` or with `PPAGE = 1` and `OFFSET = $0AABC`).

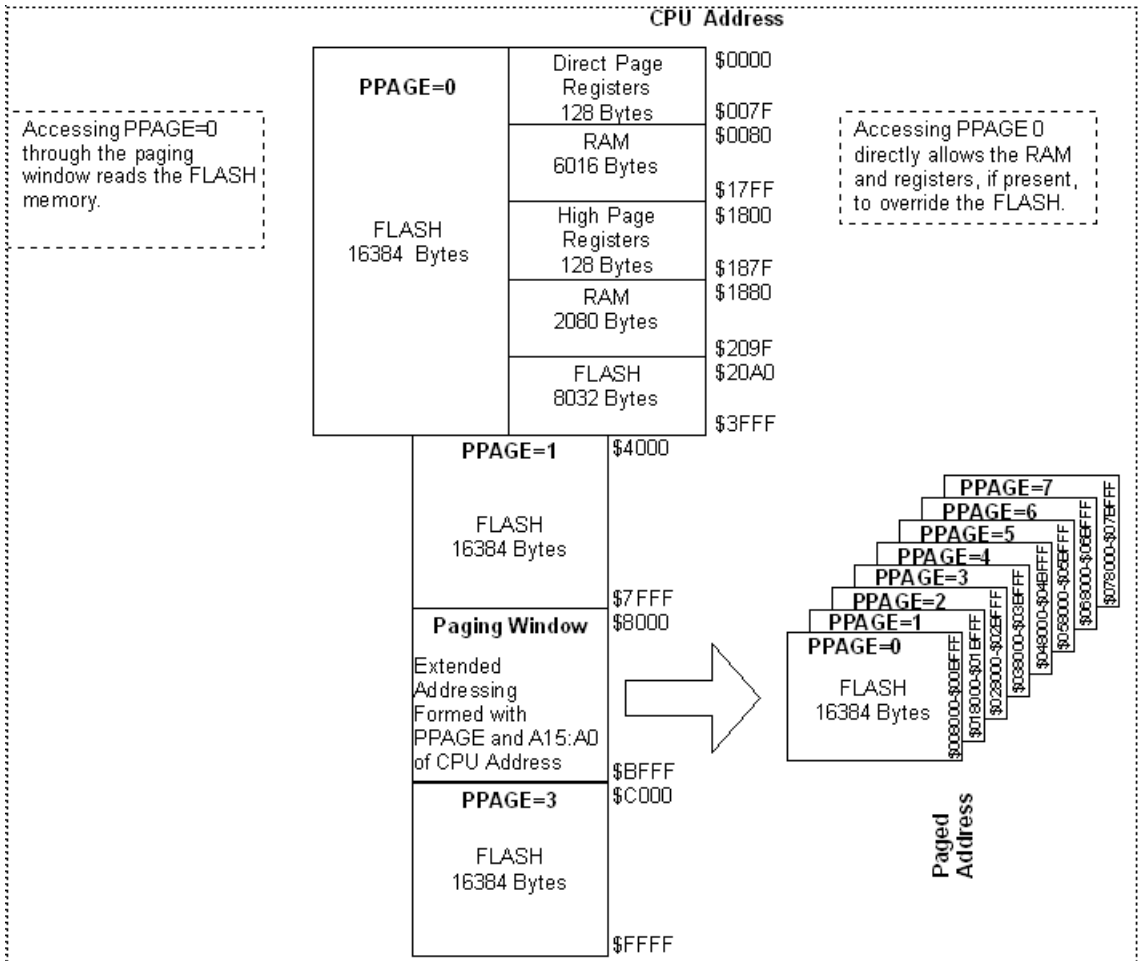

Figure 12-1. Example of Paging Mechanism

NOTE

Memory addresses may vary depending on the derivative used.

12.1.1.2 Code Banking and Linker Support

The user can enter paged addresses in the linker parameter file. The high byte of such an address is interpreted as page number. An example: 0x048100. Here, 4 is the page number, while the lower two bytes contain an address within the paging window. This particular address refers to byte 0x100 within page 4. This format does not impact non-banked address specification. The following example defines pages 4 through 7 as sections in the linker parameter file:

```
ROM_FAR_1           = READ_ONLY  0x48000 TO 0x4BFFF;
```

```
ROM_FAR_2            = READ_ONLY  0x58000 TO 0x5BFFF

ROM_FAR_3            = READ_ONLY  0x68000 TO 0x6BFFF

ROM_FAR_4             = READ_ONLY  0x78000 TO 0x7BFFF
```

12.1.2 SMALL Model

The SMALL memory model, the default, assumes that all pointers and functions have 16-bit addresses unless explicitly specified otherwise. This memory model requires code and data to be located in the 64-kilobyte address space. Use the `-Ms` compiler option to select the SMALL memory model (see -M (-Mb, -Ms, -Mt): Memory Model).

12.1.3 TINY Model

In the TINY memory model, all data including stack must fit into the zero page. This memory model assumes that data pointers have 8-bit addresses unless explicitly specified with the keyword `__far`. The code address space is 64 kilobytes and function pointers are 16-bit in size. Use the `-Mt` compiler option to select the TINY memory model.

12.2 Non-ANSI Keywords

The following table gives an overview of the supported non-ANSI keywords.

Table 12-1. Supported Non-ANSI Keywords

Keyword	Supported For		
	Data Pointer	Function Pointer	Function
__far	yes	no	no
__near	yes	no	no
interrupt	no	no	yes

12.3 Data Types

This section describes HC(S)08 Backend implementation of the basic ANSI-C types. The topics covered are as follows:

- Scalar Types
- Floating-Point Types
- Bitfields
- Pointer Types and Function Pointers
- Structured Types and Alignment
- Object Size
- Register Usage

12.3.1 Scalar Types

Use the `-T` compiler option to change any of the basic types (see *-T: Flexible Type Management*). Scalar types (except `char`) do not have a signed/unsigned qualifier, and their default values are signed (for example, `int` is the same as `signed int`).

NOTE

When using Flexible type management, use Minimal Startup Code instead of ANSI startup code.

the following table gives the sizes and possible formats of the simple types using the `-T` option.

Table 12-2. Floating-Point Representation

Type	Default Format	Default Value Range		Formats Available with the -T Option
		Min	Max	
char (unsigned)	8 bit	0	255	8 bit, 16 bit, 32 bit
signed char	8 bit	-128	127	8 bit, 16 bit, 32 bit
unsigned char	8 bit	0	255	8 bit, 16 bit, 32 bit
signed short	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
unsigned short	16 bit	0	65535	8 bit, 16 bit, 32 bit
enum (signed)	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
signed int	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
unsigned int	16 bit	0	65535	8 bit, 16 bit, 32 bit

Table continues on the next page...

Table 12-2. Floating-Point Representation (continued)

Type	Default Format	Default Value Range		Formats Available with the -T Option
		Min	Max	
signed long	32 bit	-2147483648	2147483647	8 bit, 16 bit, 32 bit
unsigned long	32 bit	0	4294967295	8 bit, 16 bit, 32 bit
signed long long	32 bit	-2147483648	2147483647	8 bit, 16 bit, 32 bit
unsigned long long	32 bit	0	4294967295	8 bit, 16 bit, 32 bit

NOTE

Default value for plain type `char` is unsigned. Use the `-T` option to change the default.

12.3.2 Floating-Point Types

The Compiler supports the two IEEE standard formats (32 and 64 bits wide) for floating-point types. The following table shows the range of values for the various floating-point representations.

The Compiler implements the default format for a `float` as 32-bit IEEE, and `double` as IEEE 64-bit format. If you need speed more than the added accuracy of double arithmetic operations, issue the -Fd: Double is IEEE32 command-line option. Using this option, the Compiler implements both float and double using the IEEE 32-bit format.

Use the -T: Flexible Type Management option to change the default format of a `float` or `double`.

Table 12-3. Floating-Point Representation

Type	Default Format	Default Value Range		Formats Available with the -T Option
		Min	Max	
float	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64
long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64
long long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64

12.3.3 Bitfields

The Compiler allows a maximum bitfield width of 16 bits, with byte-size allocation units. The Compiler uses words only when a bitfield exceeds eight bits, or when using bytes causes more than two bits to be left unused. Allo cation order is from the least significant bit up to the most significant bit in the order of declaration. The following figure illustrates an allocation scheme.

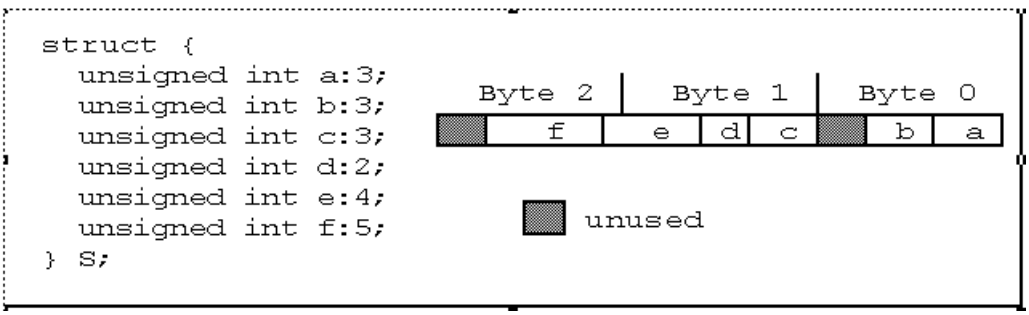

Figure 12-2. Allocation of Six Bitfields

The following example demonstrates a simple C code source with various ways to access bitfields, together with the produced code:

Listing: Demonstration of Bitfield Instructions for the HC(S)08 Compiler

```
#pragma DATA_SEG __SHORT_SEG _zpage /* place following variables into
 zero page */

#define BIT_SET(x,bitNo) ((x) |= 1<<(bitNo))

#define BIT_CLR(x,bitNo) ((x) &= ~(1<<(bitNo)))

char i, j;

struct {

  unsigned int b0:1;

  unsigned int b1:1;

} B;

void main(void) {

  /* demo using BSET/BCLR/BRSET/BRCLR */

  if (i&1) { /* BRCLR */

    /* BCLR: clearing a bit */

    i &= ~1;         /* using normal ANSI-C */

    BIT_CLR(j,0);    /* using a macro */

    __asm BCLR 0,i   /* using inline assembly */

  }
```

```
  if ((i&1) == 0) { /* BRSET */
    /* BSET: setting a bit */
    i |= 1;          /* using normal ANSI-C */
    BIT_SET(j,0);    /* using a macro */
    __asm BCLR 0,j   /* using inline assembly */
  }
  if (i&4) { /* BRCLR */
    i &= ~4; /* BCLR */
  }
  if ((i&4) == 4) { /* BRSET */
    i |= 4; /* BSET */
  }
  /* demo using bitfields in ANSI-C (warning: not portable, depends on
     the compiler
     how bitfields are allocated! */
  if (B.b0) {
    B.b1 = 1;
  } else if (B.b1) {
    B.b0 = 0;
  }
}
#if 0
   13:  void main(void) {
   14:    /* demo using BSET/BCLR/BRSET/BRCLR */
   15:    if (i&1) { /* BRCLR */
   16:      /* BCLR: clearing a bit */
00000000 010006           BRCLR  0,i,*6       /abs = 0009
   17:      i &= ~1;        /* using normal ANSI-C */
00000003 1100             BCLR   0,i
   18:      BIT_CLR(j,0);   /* using a macro */
00000005 1100             BCLR   0,j
   19:      __asm BCLR 0,i    /* using inline assembly */
   20:    }
00000007 1100             BCLR   0,i
   21:    if ((i&1) == 0) { /* BRSET */
   22:      /* BSET: setting a bit */
```

```
00000009 000006           BRSET  0,i,*6       /abs = 0012
   23:      i |= 1;         /* using normal ANSI-C */
0000000C 1000             BSET   0,i
   24:      BIT_SET(j,0);   /* using a macro */
0000000E 1000             BSET   0,j
   25:    __asm BCLR 0,j    /* using inline assembly */
   26:    }
00000010 1100             BCLR   0,j
   27:    if (i&4) { /* BRCLR */
00000012 050002           BRCLR  2,i,*2       /abs = 0017
   28:      i &= ~4; /* BCLR */
   29:    }
00000015 1500             BCLR   2,i
   30:    if ((i&4) == 4) { /* BRSET */
00000017 050002           BRCLR  2,i,*2       /abs = 001C
   31:      i |= 4; /* BSET */
   32:    }
   33:    /* demo using bitfields in ANSI-C (warning: not portable,
             depends on the compiler
   34:      how bitfields are allocated! */
0000001A 1400             BSET   2,i
   35:    if (B.b0) {
0000001C 010003           BRCLR  0,B,*3       /abs = 0022
   36:      B.b1 = 1;
0000001F 1200             BSET   1,B
00000021 81               RTS
   37:    } else if (B.b1) {
00000022 0300FC           BRCLR  1,B,*-4      /abs = 0021
   38:      B.b0 = 0;
   39:    }
00000025 1100             BCLR   0,B
   40:  }
00000027 81               RTS
#endif
```

12.3.4 Pointer Types and Function Pointers

the following table shows data pointer sizes depending on memory model and `__far`, `__near`, or `__linear` keyword usage:

Table 12-4. Data Pointer Sizes

Memory Model	Compiler Option	Default Pointer Size	__near Pointer Size	__far Pointer Size	__linear Pointer Size
SMALL	-Ms (default)	2	1	2	3
TINY	-Mt	1	1	2	3
Banked	-Mb	2	2	2	3

Function pointer size is always two bytes for SMALL and TINY memory models, and three bytes for Banked memory model.

12.3.5 Structured Types and Alignment

The Compiler does not align the following items:

- Objects allocated in memory
- Elements of an array
- struct members
- union members

The Compiler allocates local variables on the stack (which is growing downwards), and always stores the most significant part of a simple variable at the low memory address (big endian).

NOTE

The Compiler is free to align variables and fields. Always use implementation-independent access.

12.3.6 Object Size

The maximum size of an object is 32 kilobytes.

12.3.7 Register Usage

The Compiler uses all the standard registers of the HC(S)08. The stack pointer SP is used as stack pointer and as a frame pointer.

12.4 Calling Protocol and Conventions

The native HC08 and the new HCS08 have unique calling protocols. For the HCS08, the H register loads and stores more easily when loaded and stored with the X register. Thus the HCS08 uses the H register for parameter passing and return values.

The topics covered here are as follows:

- HC08 Argument Passing
- HCS08 Argument Passing (used for the -Cs08 Option)
- HC08 Return Values
- HCS08 Return Values (used for the -Cs08 Option)
- Returning Large Objects
- Stack Frames
- Pragma TRAP_PROC
- Interrupt Vector Table Allocation
- Segmentation
- Optimizations
- Volatile Objects

12.4.1 HC08 Argument Passing

The HC08 uses the C calling convention for all functions. The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

When a function contains a fixed number of arguments and the size of the last parameter is two bytes, the HC08 passes last parameter in `X` and `A`.

When a function contains a fixed number of arguments and the size of the last parameter is one byte and the size of the next to last parameter is greater than one, the HC08 passes the last parameter in A. When the size of the next to last parameter is also one byte, the HC08 passes the next to last parameter in A and the last one in X.

12.4.2 HCS08 Argument Passing (used for the -Cs08 Option)

The HCS08 uses the C calling convention for all functions. The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

When a function contains a fixed number of arguments and the size of the last parameter is two bytes, the HCS08 passes the last parameter in H and X.

When a function contains a fixed number of arguments and the size of the last parameter is one byte, the HCS08 passes the last parameter in A. If the size of the next to last parameter is also one byte, the HCS08 passes the next to last parameter in X. If the size of the next to last parameter is two bytes, the HCS08 passes the next to last parameter in H and X.

12.4.3 HC08 Return Values

The HC08 returns function results in registers, unless the function returns an object with a size greater than two bytes. The register used depends on the return type, as shown in the following table.

Table 12-5. HC08 Return Values

Return Type	Registers
char (signed or unsigned)	A
int (signed or unsigned)	X:A
pointers/arrays	X:A
function pointers	X:A

12.4.4 HCS08 Return Values (used for the -Cs08 Option)

The HCS08 returns function results in registers, unless the function returns an object with a size greater than two bytes. The register used depends on the return type, as shown in the following table.

Table 12-6. HCS08 Return Values

Return Type	Registers
char (signed or unsigned)	A
int (signed or unsigned)	H:X
Pointers or arrays	H:X
Function pointers	H:X

12.4.5 Returning Large Objects

Both the HC08 and the HCS08 call functions returning objects larger than two bytes with an additional parameter. This parameter, the address to which to copy the object, passes in `H:X`.

12.4.6 Stack Frames

Both systems use a stack frame as a database for passing parameters to and from functions that use stack in RAM.

The topics covered here are as follows:

- Frame Pointer
- Entry Code
- Exit Code

12.4.6.1 Frame Pointer

F unctions normally have a stack frame containing all their local data. The Compiler does not set up an explicit frame pointer, but generates code to access local data and parameters on the stack relative to the SP register.

12.4.6.2 Entry Code

The *entry code* reserves space for local variables (refer the following listing).

Listing: Entry Code

```
PSHA      ; Only if there are register parameters
PSHX      ; Only if there are register parameters

AIS #(-s) ; Reserved space for local variables and spills
```

In this case, `s` is the size (in bytes) of the local data of the function. There is no static link, and the dynamic link is not stored explicitly.

12.4.6.3 Exit Code

Exit code removes local variables from the stack and returns to the caller (refer the following listing).

Listing: Exit Code

```
AIS #(t)  ; Remove local stack space,
          ; including an eventual register parameter

RTS       ; Return to caller
```

12.4.7 Pragma TRAP_PROC

This pragma defines an interrupt routine (that is, activating this pragma terminates the function with an RTI instruction instead of an RTS). Normally, interrupt routines save and restore the `H` register at the entry response exit. If you are sure that `H` is not written in the interrupt routine, you can use the `TRAP_PROC` pragma to disable the saving and restoring of `H` (see #pragma TRAP_PROC: Mark Function as Interrupt Function).

12.4.8 Interrupt Vector Table Allocation

The Compiler provides a non-ANSI compliant way to directly specify the interrupt vector number in the source:

```
void
interrupt 0 ResetFunction(void) {


  /* reset handler */


}
```

The Compiler uses the translation from interrupt vector number to interrupt vector address shown in the followingthe following.

Table 12-7. Interrupt Vector Translation to Vector Address

Vector Number	Vector Address	Vector Address Size
0	0xFFFE, 0xFFFF	2
1	0xFFFC, 0xFFFD	2
2	0xFFFA, 0xFFFB	2
...	...	...
n	0xFFFF - (n*2)	2

12.4.9 Segmentation

The Linker memory space may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions, which the Linker then allocates into that segment. An entry in the Linker parameter file determines where that segment actually lies.

Three pragmas specify code segments and data segments:

```
#pragma DATA_SEG [__SHORT_SEG] <name>


#pragma CONST_SEG [__LINEAR_SEG] <name>


#pragma STRING_SEG [__LINEAR_SEG] <name>
```

All remain valid until the Compiler encounters the next pragma of the same kind. Unless you specify different segments, the Compiler assumes two default segments named DEFAULT_ROM (the default code segment) and DEFAULT_RAM (the default data segment). To explicitly set these default segments as the current segments, use the segment name DEFAULT ():

Listing: Explicit Default Segments

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
```

The additional keyword __SHORT_SEG informs the Compiler that a data segment is allocated in the zero page (address range from `0x0000` to `0x00FF`):

```
#pragma DATA_SEG __SHORT_SEG <segment_name>
```

or

```
#pragma DATA_SEG __SHORT_SEG DEFAULT
```

Using the zero page enables the Compiler to generate much denser code because it uses DIRECT addressing mode instead of EXTENDED.

NOTE

It is the programmer's responsibility to actually allocate __SHORT_SEG segments in the zero page in the Linker parameter file. For more information, see the Linker section of the Build Tools manual.

12.4.10 Optimizations

The Compiler applies a variety of code improving techniques commonly defined as `optimizations`. This section gives a short overview of the most important optimizations, the topics covered are as follows:

- Lazy Instruction Selection
- Strength Reduction
- Shift Optimizations
- Accessing Bitfields
- HC08 Branch Optimizations
- Optimization for Execution Time or Code Size

12.4.10.1 Lazy Instruction Selection

This simple optimization replaces certain instructions with shorter and/or faster equivalents. Examples include using of TSTA instead of CMP #0 or COMA instead of EORA #0xFF.

12.4.10.2 Strength Reduction

This optimization replaces expensive operations with cheaper operations, reducing either execution time or code size. Examples include using left or right shifts instead of multiplications and divisions by constant powers of two.

12.4.10.3 Shift Optimizations

Shifting a byte variable by a constant number of bits is intensively analyzed. The Compiler always tries to implement such shifts in the most efficient way. As an example, consider the following:

```
char a, b;


a = b << 4;
```

This disassembles in the following code:

```
LDA   b


NSA         ; Swap nibbles...


AND   #-16 ; ...and mask!
```

```
  STA   a
```

12.4.10.4 Accessing Bitfields

Mask and shift operations involved in packing or unpacking bitfields cause inefficient bitfield accesses. To increase efficiency when accessing bitfields only one bit wide, use direct addressing mode, the BSET/BCLR instruction, and the bit branches BRSET and BRCLR.

To enable direct addressing, declare the bitfield in the zero page using the DATA_SEG__SHORT_SEG pragma (refer the following listing).

Listing: Example

```
#pragma DATA_SEG __SHORT_SEG DATA_ZEROPAGE;
struct {

  int a:1;

} bf;

void main(void) {

  bf.a = -1;

}
```

12.4.10.5 HC08 Branch Optimizations

This optimization minimizes the span of branch instructions. The Compiler replaces a relative branch by an inverted condition branch across an unconditional jump, unless the offset of this branch is in the range [-128 to 127]:

```
    BRcc dest


    ...; More than 127 bytes of code


  dest:
```

Provided the code indicated by "..." doesn't contain another branch to label dest, the Compiler changes the previous code to:

```
    BR~cc skip

    JMP   dest

skip:

    ...    ; More than 127 bytes of code

dest:
```

Also, the Compiler may resolve branches to branches into two branches to the same target. The Compiler may remove redundant branches (for example, a branch to the instruction immediately following it).

The code above has the following effect:

1. The opcode byte of `BRN` (branch never) replaces an unconditional branch over one byte.
2. The next byte is skipped (opcode decoded as `SKIP1`).
3. The opcode of `CPHX` < `immediate_16`> replaces an unconditional branch over two bytes, if no flags are needed afterwards.
4. The following two bytes are skipped (opcode decoded as `SKIP2`).
5. One byte is gained for `SKIP1` and two bytes are gained for `SKIP2`.

The execution speed remains unchanged. Use the -OnB: Disable Branch Optimizer compiler option to disable this optimization.

12.4.10.6 Optimization for Execution Time or Code Size

At times the Compiler must choose between generating fast, but large code, or small but slower code. Usually the Compiler optimizes on code size. It often has to decide between a runtime routine and expanded code. In these cases, the Compiler chooses runtime routine only if it is at least three bytes shorter than the expanded instruction sequence.

12.4.11 Volatile Objects

The Compiler does not do register tracing on volatile global objects and does not eliminate accesses to volatile global objects.

12.5 Generating Compact Code with the HC08 Compiler

Some compiler options or use of `__SHORT_SEG` segments help you to generate compact code with the HC08 compiler. The topics covered are as follows:

- Compiler Options
- __SHORT_SEG Segments
- Defining I/O Registers

12.5.1 Compiler Options

Using the `-Cni` (non-integral promotion on integer compiler) option helps reduce the size of the code generated.

When you specify this option, ANSI C integral promotion does not apply to character comparison or arithmetic operations. This dramatically reduces the amount of code.

12.5.2 __SHORT_SEG Segments

Variables allocated on the direct page (between 0 and `0xFF`) are accessed using direct addressing mode. The Compiler allocates variables on the direct page if you define the variables with a `__SHORT_SEG` segment.

Listing: Example

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;

#pragma DATA_SEG DEFAULT

unsigned int myvar3, myVar4.
```

In the previous example, access both myVar1 and myVar2 using direct addressing mode. Access the variables myVar3 and myVar4 using extended addressing mode.

Defining some exported variables in a __SHORT_SEG segment requires that the external declaration for these variables also specify a __SHORT_SEG segment allocation.

Listing: External Definition of the Above Variable

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
extern unsigned int myVar1, myVar2;

#pragma DATA_SEG DEFAULT

extern unsigned int myvar3, myVar4.
```

Place the segment on the direct page in the PRM file.

Listing: Example

```
LINK test.abs
NAMES test.o start08.o ansi.lib END

SECTIONS

    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;

    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;

    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;

PLACEMENT

    DEFAULT_ROM                    INTO  MY_ROM;

    DEFAULT_RAM                    INTO  MY_RAM;

    _DATA_ZEROPAGE, myShortSegment INTO  Z_RAM;

END

STACKSIZE 0x60

VECTOR ADDRESS 0xFFFE _Startup /* set reset vector on _Startup */
```

NOTE

The linker is case-sensitive. The segment name must be identical in the C and PRM files.

If all data and stack fits into the zero page, use the tiny memory model for convenience.

12.5.3 Defining I/O Registers

The HC(S)08 I/O Registers are usually based at address 0. To tell the compiler to use direct addressing mode to access the I/O registers, define these registers in a __SHORT_SEG section based at the specified address.

In the C source file, define the I/O registers as follows:

Listing: I/O Example

```
typedef struct {
  unsigned char SCC1;

  unsigned char SCC2;

  unsigned char SCC3;

  unsigned char SCS1;

  unsigned char SCS2;

  unsigned char SCD;

  unsigned char SCBR;

} SCIStruct;

#pragma DATA_SEG __SHORT_SEG SCIRegs

volatile SCIStruct   SCI;

#pragma DATA_SEG DEFAULT
```

Place the segment at the appropriate address in the PRM file.

Listing: Example PRM File

```
LINK test.abs
NAMES test.o start08.o ansi.lib END

SECTIONS

    SCI_RG = READ_WRITE  0x0013 TO 0x0019;

    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;

    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;

    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;

PLACEMENT

    DEFAULT_ROM                   INTO  MY_ROM;

    DEFAULT_RAM                   INTO  MY_RAM;

    _DATA_ZEROPAGE                INTO  Z_RAM;

    SCIRegs                       INTO  SCI_RG;

END

STACKSIZE 0x60

VECTOR ADDRESS 0xFFFE _Startup /* set reset vector on _Startup */
```

NOTE

The linker is case-sensitive. The segment name must be identical in the C and PRM files.

Chapter 13
High-Level Inline Assembler for the HC(S)08

The High-Level Inline (HLI) Assembler makes full use of the target processor properties from within a C program. There is no need to write a separate assembly file, assemble it, and later bind it with the rest of the application written in ANSI-C with the inline assembler. The Compiler does all the work. For detailed information, refer to the Freescale HC(S)08 Family Reference Manual.

This chapter lists the following topics:

13.1 Syntax

I nline assembly statements can appear anywhere a C statement can appear (an __asm statement must be inside a C function). Inline assembly statements take one of two forms, which have various configurations. Using the first form, you can put multiple __asm statements on one line and delimit comments like regular C or C++ comments. Using the second form, you can contain one to several assembly instructions within the __asm block, but only one assembly instruction per line is possible. Also, the semicolon starts an assembly comment.

```
"__asm" <Assembly Instruction> ";" ["/*" Comment "*/"]


"__asm" <Assembly Instruction> ";" ["//" Comment]
```

or

```
"__asm" "{"

 { <Assembly Instruction> [";" Comment] "\n" }

"}"
```

or

```
"__asm" " ("  <Assembly Instruction> [";" Comment] ")" ";"
```

or

```
"__asm" [" (" ] <string Assembly instruction > [")"] [";"]

with   <string Assembly instruction >

=     <Assembly Instruction>  [";" <Assembly instruction>]
```

or

```
"#asm"

<Assembly Instruction> [";" Comment] "\n"

"#endasm"
```

13.2 C Macros

C macros expand inside inline assembler code just like they expand in C. One special point to note is the syntax of the `__asm` directive (generated by macros). As macros always expand to one single line, you can only use `__asm NOP;` the first form of the __asm keyword, in macros.

for example,

```
#define SPACE_OK { __asm NOP; __asm NOP; }
```

Using the second form is illegal:

```
#define NOT_OK { __asm { \

                    NOP; \

                     NOP; \

                }
```

Here the preprocessor expands the `NOT_OK` macro to one single line, which is mistranslated because every assembly instruction must be explicitly terminated by a new line.

To use `#` inside macros to build immediates, use the #pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing.

13.3 Inline Assembly Language

The following listing is an inline assembly instruction in the EBNF Syntax. A short discussion on semantics and special constructs follows.

Listing: Inline Assembly Listing in EBNF Format

```
AsmLine         = [Label](Code | Directive).
Label           = ident :.

Code            = mnemonic [ArgList][OptParams].

Directive       = Code.

ArgList         = Argument {, Argument}.
```

```
Argument       = Imm | Expression.
Imm            = # Expression.
OptParams      = ! RegList , RegList.
RegList        = { [Reg] {, Reg} }.
Reg            = A|HX|H|X|SR.
Variable       = Identifier.
Expression     = [Term {(+|-) Term}] [, (X|X+|HX|SP)].
Term           = Factor {(*|/) Factor}.
Factor         = ( Expression )
                 | - Factor
                 | @ Factor
                 | * Factor
                 | Factor {: Factor|MSB)}
                 | Number
                 | TypeQualifier
                 | Variable {. Field}
                 | Procedure
                 | Label
TypeQualifier = Type . Field {. Field}.
Procedure      = Identifier.
Label          = Identifier.
Type           = Identifier.
Field          = Identifier.
```

To resolve grammar ambiguities, take the longest possible match to reduce to Factor (shift is always preferred). The pseudo offset MSB designates the most significant byte of an effective address and is only allowed for memory operands. The TypeQualifier production above evaluates to the offset of the given field. Type must be the name of a struct type. All named fields must exist.

This section covers the following topic:

- Register Indirect Addressing Mode

13.3.1 Register Indirect Addressing Mode

The offset must be constant for the register indirect addressing modes. This constant may be an immediate or a variable address relocated by the linker.

Example:

```
  STA   @Variable_Name,X
```

The following simple example illustrates the use of the HLI Assembler:

Assuming that `str` points to a character array, you can write a simple function in assembly language to determine the length of a string:

Listing: strlen() Definition

```
int strlen (char *str)
  /*** The 'str' character array is passed on the stack. strlen returns
       length of 'str'.

       This procedure assumes len(str) is smaller than 256! */

{

  __asm {

    LDHX str        ; load pointer

    CLRA            ; init counter

    BRA test        ; go to test

  loop:

    AIX   #1        ; increment pointer

    INCA            ; increment counter

  test:

    TST   0,X       ; not end of string?

    BNE   loop      ; next char

    CLRX            ; return value in X:A(see later)

  };

  /* C statements could follow here */

}
```

NOTE

Unless `#pragma NO_ENTRY` is set, the Compiler takes care of entry and exit code. You do not have to worry about setting up a stack frame.

13.4 Special Features

The remainder of this section covers special HLI features for the HC(S)08. This section covers the following topic:

- Caller/Callee Saved Registers
- Reserved Words
- Pseudo Opcodes
- Accessing Variables
- Address Notation
- H:X Instructions
- Constant Expressions
- Optimizing Inline Assembly
- Assertions
- Stack Adjust
- In and Gen Sets

13.4.1 Caller/Callee Saved Registers

Because the compiler does not save any registers on the caller/callee side, you do not have to save or restore any registers in the HLI across function calls.

13.4.2 Reserved Words

The inline assembler knows some reserved words which must not collide with user-defined identifiers such as variable names. These reserved words are:

- All opcodes (LDA, STX, ...)
- All register names (A, H, X, HX, SR, SP)
- The identifier MSB

These reserved words are not case-sensitive in the inline assembler; that is, LDA is the same as lda or even LdA. For all other identifiers (labels, variable names and so on), the inline assembler is case-sensitive.

13.4.3 Pseudo Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

- DCB 1 ; Byte constant 1
- DCB 0 ; Byte constant 0
- DCW 12 ; Word constant 12
- DCL 20, 23 ; Longword constants
- DCF 1.85 ; IEEE 32bit float
- DCD 2.0 ; double IEEE64

13.4.4 Accessing Variables

The inline assembler allows accessing local and global variables declared in C. Simply use the variable names in the instruction.

13.4.5 Address Notation

In addition to the address syntax described in the HC(S)08 Family Manual, one may access objects using syntax similar to C and symbolic variable names. In this case all addresses are assumed to be of type char *.

```
@var       ==> address of var


*var       ==> object pointed to by var


var:offs   ==> *((@var) + offs)


obj.field  ==> *((@obj) + offset_of(field))
```

```
type.field ==> offset_of(field)
```

13.4.6 H:X Instructions

The Compiler allows LDHX with any reasonable argument and does not restrict it to DIRECT page and IMMEDIATE arguments. In the native HC08, `LDHX` instructions represent code patterns which translate automatically into an appropriate code sequence, unless the option `-cs08` is active. Consider the following example (assume local is a variable on the stack).

Table 13-1. H:X Instructions

Inline Assembly	Expansion
LDHX local	LDX local:0 PSHX PULH LDX local:1
LDHX @local	TSX AIX #<local-offset>

13.4.7 Constant Expressions

Use constant expressions anywhere an IMMEDIATE value is expected, to contain the binary operators for addition (+), subtraction (-), multiplication (*), and division (/). Also, the unary operator (-) is allowed. Use round brackets to force an evaluation order other than the normal one. The number syntax is the same as in ANSI-C.

You may also use a constant expression as the address of a global variable or the offset of a local variable.

13.4.8 Optimizing Inline Assembly

The Compiler is free to modify the instruction stream as long as its semantics remains the same. This is especially true for stack pointer indirect accesses (ind, SP, for example, accesses to local variables), which may be replaced by TSX and a ind, H:X access.

13.4.9 Assertions

The pseudo instruction below declares that the register <reg> (one of H, X, HX, A) contains addr at this inst ruction.

```
_ASSERT  <reg>, addr
```

The Compiler may use this instruction to perform optimizations. The code is correct even if the Compiler ignores this information. The following listing shows an example.

Listing: Using _ASSERT

```
    LDHX  @a:4
    MOV   #4, tmp

loop:

    AIX   #-1

    LDA   0,X

    STA   4,X

    DBNZ  tmp, loop

    _ASSERT HX, @a
```

13.4.10 Stack Adjust

ICG-based versions of the compiler no longer support the `_STACK` pseudo instruction. For example:

```
label:


  _STACK  #offs
```

This pseudo instruction declares that the stack pointer's value at this instruction is (stack pointer's value at label) `+ offs`. It is ignored by the ICG-based Compiler. Also:

```
_ADJ  #offs


label:
```

This pseudo instruction adjusts the SP register with appropriate AIS instructions, so that _STACK holds (value at label+offs) afterwards. The _STACK instruction is useful for situations like the following:

Listing: Using the _STACK Pseudo Instruction

```
base:
    CLRA

    MOV   #8, tmp

loop:

    PSHA

    DBNZ  tmp, loop

    _STACK #-8      ; The pseudo instruction

_STACK  label:offs
```

13.4.11 In and Gen Sets

The pseudo instruction below announces to the Compiler that registers in <in-set> are used further, and those in <gen-set> are not.

```
_INGEN  <in-set>, <gen-set>
```

The notation below declares which registers are used and which ones are modified by an instruction:

```
<instr> <args> ! <in-set>, <gen-set>
```

This works well for JSR, JMP, and RTS (and similar) instructions. The default is always on the conservative side. Consider the following example.

Listing: Example Program

```
    LDA a:1
    ADD b:1

    STA b:1

    LDA a:0

    ADC b:0

    STA b:0

    RTS ! {}, {}
```

This allows the Compiler to discard the H:X register for accesses to local variables. With the simple RTS, the Compiler does not know if the H:X register is to be preserved or not. Separate multiple registers with a +. For example:

```
RTS ! {A+X},  {}
```

13.4.11.1 Getting the High-Address Part in HLI

You can use the MSB syntax to get the high-address part of an object. Consider the following example.

Listing: Getting the High-Address Part of an Object

```
int i, *p_i;
void main(void) {

  /* p_i = &i */

  __asm {

          LDA @i

          STA p_i:1

          LDA @i:MSB

          STA p_i

}
```

The following examples illustrate the usage with the original code and the generated one.

Listing: Four Examples of HLI Usage and the Generated Assembly Code

```
char f(void);
char asm1(int a) {

  int b;

  __asm {

    LDA   a:1

    STA   b:1

    JSR   f ! {}, {}

    STA   a:0

    LDA   b:1

    RTS   ! {}, {A}

  }

}
```

```
char f(void);
char asm1(int a) {

  int b;

  __asm {

  TSX

  LDA 5,X  ; H:X is used

  STA 1,X  ; to access all locals

  JSR f

  STA 4,X

  LDA 1,X

  RTS

}
```

```
                          }
char asm2(int a) {        char asm2(int a) {
  int b;                    int b;
  __asm {                   __asm {
    LDA   a:1                 LDA 6,SP ; X is input register
    STA   b:1                 STA 2,SP ; for f. SP must be used
    JSR   f ! {X}, {}         JSR f
    STA   a:0                 TSX      ; f does not set X.
    LDA   b:1                 STA 4,X  ; now X is free
    RTS   ! {}, {A}           LDA 1,X
  }                             RTS
}                           }
                          }
char asm3(int a) {        char asm3(int a) {
  int b;                    int b;
  __asm {                   __asm {
    LDA   a:1                 TSX
    STA   b:1                 LDA 5,X   ; H:X is used
    JSR   f ! {}, {X}         STA 1,X   ; to access locals
    STA   a:0                 JSR f     ; f destroys X
    LDA   b:1                 TSX       ; X is restored
    RTS   ! {}, {A}           STA 4,X   ; to access locals
  }                           LDA 1,X
}                             RTS
                            }
                          }
char asm4(int a) {        char asm4(int a) {
  int b;                    int b;
  __asm {                   __asm {
    LDA   a:1                 LDA 6,SP
    STA   b:1                 STA 2,SP
    JSR   f ! {}, {}          JSR f
    STA   a:0                 STA 5,SP
    LDA   b:1                 LDA 2,SP
    RTS   ! {X}, {A}          RTS ; X must not be used
```

```
                                    ; in this function!
  }                         }
}                           }
```

Chapter 14
ANSI-C Library Reference

This section covers the ANSI-C Library.

- Library Files : Describes the types of library files
- Special Features : Describes the special considerations of the ANSI-C standard library relating to embedded systems programming
- Library Structure : Examines the various elements of the ANSI-C library, grouped by category
- Types and Macros in the Standard Library : Discusses all types and macros defined in the ANSI-C standard library
- The Standard Functions : Describes all functions in the ANSI-C library

Chapter 15
Library Files

This chapter covers the following topics:

15.1 Directory Structure

The library files are delivered in the following structure.

Listing: Layout of Files after a CodeWarrior IDE Installation

```
<install>\lib\<target>c\        /* readme files, make files */
<install>\lib\<target>c\src     /* C library source files */

<install>\lib\<target>c\include /* library include files */

<install>\lib\<target>c\lib     /* default library files */

<install>\lib\<target>c\prm     /* Linker parameter files */
```

Check out the README.TXT located in the library folder for additional information on memory models and library filenames.

15.2 Generating a Library

In the directory structure above, a CodeWarrior * `.mcp` file can build all the libraries and the startup code object files. Simply load the `<target>_lib.mcp` file into your CodeWarrior project and build all the targets.

15.3 Common Source Files

The following table lists the target-independent source and header files of the Standard ANSI Library.

Table 15-1. Standard ANSI Library-Target Independent Source and Header Files

Source File	Header File
alloc.c	
assert.c	assert.h
ctype.c	ctype.h
	errno.h
heap.c	heap.h
	limits.h
math.c, mathf.c	limits.h, ieemath.h, float.h
printf.c, scanf.c	stdio.h
signal.c	signal.h
	stdarg.h
	stddef.h
stdlib.c	stdlib.h
string.c	string.h
	time.h

15.4 Startup Files

Because every memory model needs special startup initialization, the library files also contain startup object files compiled with different Compiler option settings (see Compiler Options for details).

Depending on the memory model chosen, link the correct startup file with the application. The floating point format used does not matter for the startup code.

Note that the library files contain a generic startup written in C as an example of doing all the tasks needed for a startup:

- Zero Out
- Copy Down

- Register initialization
- Handling ROM libraries

Because not all of the above tasks may be needed for an application and for efficiency reasons, special startup is provided as well (e.g., written in HLI). However, you can also use the version written in C as well. For example, compile the `startup.c` file with the memory/options settings and link it to the application.

The topics covered here are as follows:

- Startup Files for HC08
- Startup Files for HCS08

15.4.1 Startup Files for HC08

Depending on the memory model, link a different startup object file to the application. The following table lists the startup files for the HC08.

Table 15-2. Startup Files for HC08

Startup Object File	Startup Source File	Compiler Options
start08.o	Start08.c	-Ms
start08p.o	Start08.c	-Ms -C++f
start08t.o	Start08.c	-Mt
start08tp.o	Start08.c	-Mt -C++f

15.4.2 Startup Files for HCS08

Depending on the memory model, link a different startup object file to the application. The following table lists the startup files for the HCS08:

Table 15-3. Startup Files for HCS08

Startup Object File	Startup Source File	Compiler Options
start08s.o	Start08.c	-Ms -Cs08
start08ts.o	Start08.c	-Mt -Cs08

15.5 Library Files

NOTE

This feature is not yet implemented in the current release of the MCU V10.x tools.

The CodeWarrior IDE delivers most of the ANSI library object files in the form of an object library (see below).

Several Library files are bundled with the Compiler. Different memory models or floating point formats require different library files.

The library files contain all necessary runtime functions used by the compiler and the ANSI Standard Library as well. The list files (`*.lst` extension) contain a summary of all objects in the library file.

To link against a modified file which also exists in the library, specify the file first in the link order.

Check the `readme.txt` located in the library structure (`lib\<target>c\README.TXT`) for a list of all delivered library files and the related memory models or options used.

Chapter 16
Special Features

Not everything defined in the ANSI standard library relates to embedded systems programming. Therefore, some functions remain unimplemented, and some have been left open for implementation because they strongly depend on the actual setup of the target system.

NOTE

All unimplemented functions do a `HALT` when called. All functions are re-entrant, except rand() and srand() , because these use a global variable to store the seed, which might give problems with light-weight processes. strtok() also uses a global variable, based on the ANSI standard definition.

This chapter describes and explains these functions:

- Memory Management - malloc(), free(), calloc(), realloc(); alloc.c, and heap.c
- Signals - signal.c
- Multiple-Byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c
- Program Termination - abort(), exit(), atexit(); stdlib.c
- I/O - printf.c
- Locales - locale.*
- ctype
- String Conversions - strtol(), strtoul(), strtod(), and stdlib.c

16.1 Memory Management - malloc(), free(), calloc(), realloc(); alloc.c, and heap.c

File `alloc.c` provides a full implementation of these functions. You must specify heap location, heap size, and heap overrun processes.

Address all of these points in the heap.c file. View the heap as a large array with a default error handling function. Feel free to modify this function or the size of the heap to suit the needs of the application. Define the heap size in libdefs.h, LIBDEF_HEAPSIZE.

16.2 Signals - signal.c

Signals are implemented as traps. This means function signal() allows you to set a vector to some function of your own (ideally a TRAP_PROC), while function raise() is unimplemented. To ignore a certain signal, install a default handler that does nothing.

16.3 Multiple-Byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c

Because the compiler does not support multiple-byte characters, all multiple-byte routines in stdlib.c are unimplemented. If these functions are needed, the programmer must specifically write them.

16.4 Program Termination - abort(), exit(), atexit(); stdlib.c

Because programs in embedded systems usually are not expected to terminate, we only provide a minimum implementation of the first two functions, while atexit() is not implemented at all. Both abort() and exit() perform a HALT.

16.5 I/O - printf.c

The printf() library function is unimplemented in the current version of the library sets in the ANSI libraries, but is implemented in the terminal.c file. This was done because often no terminal is available or the terminal highly dependent on user hardware.

The ANSI library contains several functions which simplify printf() function implementation, with all its special cases.

Using the first, ANSI-compliant way, allocate a buffer and then use the `vsprintf()` ANSI function (refer the following listing) .

Listing: Implementation of the printf() Function

```
int printf(const char *format, ...) {
  char outbuf[MAXLINE];

  int i;

  va_list args;

  va_start(args, format);

  i = vsprintf(outbuf, format, args);

  va_end(args);

  WriteString(outbuf);

  return i;

}
```

The value of `MAXLINE` defines the maximum size of any value of `printf()`. Assume the `WriteString()` function writes one string to a terminal. The disadvantages of this solution include:

- A buffer is needed, which may use a large amount of RAM.
- No matter how large the buffer (`MAXLINE`) is, buffer overflow is always possible.

Therefore we do not recommend this solution. Newer library version provide two non-ANSI functions (`vprintf()` and `set_printf()`) to avoid both disadvantages. Because these functions are non-ANSI extensions, they are not contained in the `stdio.h` header file. Therefore, specify their prototypes before using. Refer the following listing.

Listing: Prototypes of vprintf() and set_printf()

```
int vprintf(const char *pformat, va_list args);
void set_printf(void (*f)(char));
```

The `set_printf()` function installs a callback function, which is called later for every character `vprintf()` prints.

Be advised that calls to `set_printf()` and `vprintf()` also implement the standard ANSI C `printf()` derivatives functions sprintf() and `vsprintf()`. This allows code sharing across all `printf` derivatives.

There is one limitation of the printf() implementation. Because the callback function is not passed as an argument to `vprintf()`, but held in a global variable, none of the `printf()` derivatives are reentrant. Calls to different derivatives at the same time are not allowed.

The following listing shows a simple implementation of a `printf()` with `vprintf()` and `set_printf()`.

Listing: Implementation of printf() with vprintf() and set_printf()

```
int printf(const char *format, ...){
  int i;

  va_list args;

  set_printf(PutChar);

  va_start(args, format);

  i = vprintf(format, args);

  va_end(args);

  return i;

}
```

Assume the `PutChar()` function prints one character to the terminal.

Another remark must be made about the `printf()` and `scanf()` functions. `printf.c` provides the full source code of all `printf()` derivatives and `scanf.c` provides the full source code of all `scanf() derivatives`. Specific applications usually do not use many of the features of `printf()` and `scanf()`. The source code of the `printf` and `scanf` library modules contains switches (defines) to allow the user to switch off unused parts of the code. This includes the large floating-point portions of `vprintf()` and `vsscanf()`.

16.6 Locales - locale.*

This fun ction has not been implemented.

16.7 ctype

`ctype` contains two sets of implementations for all functions. The standard is a set of macros which translate into lookup table accesses.

This table uses 257 bytes of memory, so the library provides an implementation using real functions. These functions are accessible only when the macros are undefined. After `#undef isupper`, `isupper` translates into a call to function `isupper()`. Without `undef`, the corresponding macro replaces `isupper`.

Using the functions instead of the macros saves RAM and code size, at the expense of some additional function call overhead.

16.8 String Conversions - strtol(), strtoul(), strtod(), and stdlib.c

ANSI standards for string conversions require that range checking be done. Set the variable `errno` accordingly and special limit values are returned. The macro `ENABLE_OVERFLOW_CHECK` is set to 1 by default. To reduce code size, we recommend that you switch this macro off (clear `ENABLE_OVERFLOW_CHECK` to 0).

Chapter 17
Library Structure

This section examines the various parts of the ANSI-C standard library, grouped by category. This library not only contains a rich set of functions, but also numerous types and macros.

This chapter lists the following topics:

17.1 Error Handling

Use the global variable `errno` to do ANSI library error handling. Library routines set `errno` and it may be tested by user programs. There also are a few functions for error handling:

```
void  assert(int expr);
```

```
void  perror(const char *msg);


char * strerror(int errno);
```

17.2 String Handling Functions

Strings in ANSI-C are always null-terminated character sequences. The ANSI library provides the following functions to manipulate such strings.

Listing: ANSI-C String Manipulation Functions

```
size_t strlen(const char *s);
char * strcpy(char *to, const char *from);

char * strncpy(char *to, const char *from, size_t size);

char * strcat(char *to, const char *from);

char * strncat(char *to, const char *from, size_t size);

int    strcmp(const char *p, const char *q);

int    strncmp(const char *p, const char *q, size_t size);

char * strchr(const char *s, int ch);

char * strrchr(const char *s, int ch);

char * strstr(const char *p, const char *q);

size_t strspn(const char *s, const char *set);

size_t strcspn(const char *s, const char *set);

char * strpbrk(const char *s, const char *set);

char * strtok(char *s, const char *delim);
```

17.3 Memory Block Functions

Closely related to the string handling functions are those operating on memory blocks. Memory block functions operate on any block of memory, whether it is null-terminated or not. Give the length of the block as an additional parameter. Also, these functions work with `void` pointers instead of `char` pointers (refer the following listing).

Listing: ANSI-C Memory Block Functions

```
void * memcpy(void *to, const void *from, size_t size);
void *
memmove(void *to, const void *from, size_t size);

int    memcmp(const void *p, const void *q, size_t size);

void * memchr(const void *adr, int byte, size_t size);

void * memset(void *adr, int byte, size_t size);
```

17.4 Mathematical Functions

The ANSI library co ntains a variety of floating point functions. The standard interface, which is defined for type `double` (*Listing: ANSI-C Double-Precision Mathematical Functions*), has been augmented by an alternate interface (and implementation) using type `float` (*Listing: ANSI-C Single-Precision Mathematical Functions*).

Listing: ANSI-C Double-Precision Mathematical Functions

```
double acos(double x);
double asin(double x);

double atan(double x);

double atan2(double x, double y);

double ceil(double x);

double cos(double x);

double cosh(double x);

double exp(double x);

double fabs(double x);

double floor(double x);

double fmod(double x, double y);

double frexp(double x, int *exp);

double ldexp(double x, int exp);

double log(double x);

double log10(double x);

double modf(double x, double *ip);

double pow(double x, double y);

double sin(double x);

double sinh(double x);

double sqrt(double x);

double tan(double x);
```

```
double tanh(double x);
```

The `float` type functions use the same names with an appended `f`.

Listing: ANSI-C Single-Precision Mathematical Functions

```
float acosf(float x);
float asinf(float x);

float atanf(float x);

float atan2f(float x, float y);

float ceilf(float x);

float cosf(float x);

float coshf(float x);

float expf(float x);

float fabsf(float x);

float floorf(float x);

float fmodf(float x, float y);

float frexpf(float x, int *exp);

float ldexpf(float x, int exp);

float logf(float x);

float log10f(float x);

float modff(float x, float *ip);

float powf(float x, float y);

float sinf(float x);

float sinhf(float x);

float sqrtf(float x);

float tanf(float x);

float tanhf(float x);
```

In addition, the ANSI library also defines some functions operating on integral values (refer the following listing).

Listing: ANSI Functions with Integer Arguments

```
int    abs(int i);
div_t  div(int a, int b);

long   labs(long l);

ldiv_t ldiv(long a, long b);
```

Furthermore, the ANSI-C library contains a simple pseudo random-number generator (refer the following listing) and a function using a seed to start the random-number generator.

Listing: Random-Number Generator Functions

```
int  rand(void);
void srand(unsigned int seed);
```

17.5 Memory Management

To allocate and deallocate memory blocks, the ANSI library provides the following functions:

```
void* malloc(size_t size);


void* calloc(size_t n, size_t size);


void* realloc(void* ptr, size_t size);


void free(void* ptr);
```

Because it is not possible to implement these functions in a way that suits all possible target processors and memory configurations, all these functions are based on the system module `heap.c` file, which can be modified by the user to fit a particular memory layout.

17.6 Searching and Sorting

The ANSI library contains both a generalized s earching and a generalized sorting procedure:

```
void* bsearch(const void *key, const void *array,


              size_t n, size_t size, cmp_func f);
```

```
void  qsort(void *array, size_t n, size_t size, cmp_func f);
```

17.7 Character Functions

These functions test or convert characters. All these functions are implemented both as macros and as functions, and, by default, the macros are active. To use the corresponding function, you have to `#undefine` the macro.

Listing: ANSI-C Character Functions

```
int isalnum(int ch);
int isalpha(int ch);

int iscntrl(int ch);

int isdigit(int ch);

int isgraph(int ch);

int islower(int ch);

int isprint(int ch);

int ispunct(int ch);
int isspace(int ch);
int isupper(int ch);
int isxdigit(int ch);
int tolower(int ch);
int toupper(int ch);
```

The ANSI library also defines an interface for multiple-byte and wide characters. The implementation only offers minimal support for this feature: the maximum length of a multiple-byte character is one byte.

```
int    mblen(char *mbs, size_t n);


size_t mbstowcs(wchar_t *wcs, const char *mbs, size_t n);


int    mbtowc(wchar_t *wc, const char *mbc, size_t n);


size_t wcstombs(char *mbs, const wchar_t *wcs size_t n);


int    wctomb(char *mbc, wchar_t wc);
```

17.8 System Functions

The ANSI standard includes some system functions for raising and responding to signals, non-local jumping, and so on.

Listing: ANSI-C System Functions

```
void      abort(void);
int       atexit(void(* func) (void));

void      exit(int status);

char*     getenv(const char* name);

int       system(const char* cmd);

int       setjmp(jmp_buf env);

void      longjmp(jmp_buf env, int val);

_sig_func signal(int sig, _sig_func handler);

int       raise(int sig);
```

To process variable length argument lists, the ANSI library provides the following functions, implemented as macros:

```
void va_start(va_list args, param);


type va_arg(va_list args, type);


void va_end(va_list args);
```

17.9 Time Functions

The ANSI library provides several fun ctions to get the current time. In an embedded systems environment, implementations for these functions cannot be provided because different targets may use different ways to count the time.

Listing: ANSI-C Time Functions

```
clock_t     clock(void);
time_t      time(time_t *time_val);

struct tm * localtime(const time_t *time_val);
```

```
time_t      mktime(struct tm *time_rec);

char      * asctime(const struct tm *time_rec);

char        ctime(const time *time_val);

size_t     strftime(char *s, size_t n,

                    const char *format,

                      const struct tm *time_rec);

double      difftime(time_t t1, time_t t2);

struct tm * gmtime(const time_t *time_val);
```

17.10 Locale Functions

These functions handle loca les. The ANSI-C library only supports the minimal C environment (refer the following listing).

Listing: ANSI-C Locale Functions

```
struct lconv *localeconv(void);
char         *setlocale(int cat, const char *locale);

int           strcoll(const char *p, const char *q);

size_t        strxfrm(const char *p, const char *q, size_t n);
```

17.11 Conversion Functions

The following listing shows functions for converting strings to numbers.

Listing: AN<codeph>SI-C String/Number Conversion Functions</codeph>

```
int atoi(const char *s);
long atol(const char *s);
double atof(const char *s);
long strtol(const char *s, char **end, int base);
unsigned long strtoul(const char *s, char **end, int base);
double        strtod(const char *s, char **end);
```

17.12 printf() and scanf()

More conversions are possible for the C functions for reading and writing formatted data. The following listing shows these functions.

Listing: ANSI-C Read and Write Functions

```
int sprintf(char *s, const char *format, ...);
int vsprintf(char *s, const char *format, va_list args);

int sscanf(const char *s, const char *format, ...);
```

17.13 File I/O

The ANSI-C library contains a fairly large file I/O interface. In microcontroller applications however, one usually does not need file I/O. In the few cases where one might need it, the implementation depends on the actual target system setup. Therefore, it is impossible for Freescale to provide an implementation for these features.

Listing: ANSI-C File I/O Functions listing contains file I/O functions while *Listing: ANSI-C Functions for Writing and Reading Characters* shows character reading and writing functions. *Listing: ANSI-C Functions for Reading and Writing Blocks of Data* shows data block reading and writing functions. *Listing: ANSI-C Formatted I/O Functions on Files* shows functions for formatted I/O on files and Listing: ANSI-C Positioning Functions shows functions for positioning data within files.

Listing: ANSI-C File I/O Functions

```
FILE* fopen(const char *name, const char *mode);
FILE* freopen(const char *name, const char *mode, FILE *f);

int   fflush(FILE *f);

int   fclose(FILE *f);

int   feof(FILE *f);

int   ferror(FILE *f);

void  clearerr(FILE *f);

int   remove(const char *name);

int   rename(const char *old, const char *new);

FILE* tmpfile(void);

char* tmpnam(char *name);

void  setbuf(FILE *f, char *buf);

int   setvbuf(FILE *f, char *buf, int mode, size_t size);
```

Listing: ANSI-C Functions for Writing and Reading Characters

```
int   fgetc(FILE *f);
char* fgets(char *s, int n, FILE *f);

int   fputc(int c, FILE *f);

int   fputs(const char *s, FILE *f);

int   getc(FILE *f);

int   getchar(void);

char* gets(char *s);

int   putc(int c, FILE *f);

int   puts(const char *s);

int   ungetc(int c, FILE *f);
```

Listing: ANSI-C Functions for Reading and Writing Blocks of Data

```
size_t fread(void *buf, size_t size, size_t n, FILE *f);
size_t fwrite(void *buf, size_t size, size_t n, FILE *f);
```

Listing: ANSI-C Formatted I/O Functions on Files

```
int fprintf(FILE *f, const char *format, ...);
int vfprintf(FILE *f, const char *format, va_list args);

int fscanf(FILE *f, const char *format, ...);

int printf(const char *format, ...);

int vprintf(const char *format, va_list args);

int scanf(const char *format, ...);
```

Listing: ANSI-C Positioning Functions

```
int  fgetpos(FILE *f, fpos_t *pos);
int  fsetpos(FILE *f, const fpos_t *pos);

int  fseek(FILE *f, long offset, int mode);

long ftell(FILE *f);

void rewind(FILE *f);
```

Chapter 18
Types and Macros in the Standard Library

This section discusses all types and macros defined in the ANSI standard library. The following header files are covered here:

- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stddef.h
- stdio.h
- stdlib.h
- time.h
- string.h
- assert.h
- stdarg.h
- ctype.h

18.1 errno.h

This header file declares two constants used as error indicators in the global variable `errno`.

```
extern int errno;

#define EDOM    -1

#define ERANGE  -2
```

18.2 float.h

This header file defines constants describing the properties of floating-point arithmetic. See the following tables.

Table 18-1. Rounding and Radix Constants

Constant	Description
FLT_ROUNDS	Gives the rounding mode implemented
FLT_RADIX	The base of the exponent

All other constants are prefixed by either FLT_, DBL_ or LDBL_. FLT_ is a constant for type float, DBL_ for double, and LDBL_ for longdouble.

Table 18-2. Other Constants Defined in float.h

Constant	Description
DIG	Number of significant digits.
EPSILON	Smallest positive x for which 1.0 + x != x.
MANT_DIG	Number of binary mantissa digits.
MAX	Largest normalized finite value.
MAX_EXP	Maximum exponent such that FLT_RADIXMAX_EXP is a finite normalized value.
MAX_10_EXP	Maximum exponent such that 10MAX_10_EXP is a finite normalized value.
MIN	Smallest positive normalized value.
MIN_EXP	Smallest negative exponent such that FLT_RADIXMIN_EXP is a normalized value.
MIN_10_EXP	Smallest negative exponent such that 10MIN_10_EXP is a normalized value.

18.3 limits.h

Defines some constants for the maximum and minimum allowed values for certain types. Refer the following table.

Table 18-3. Constants Defined in limits.h

Constant	Description
CHAR_BIT	Number of bits in a character
SCHAR_MIN	Minimum value for signed char
SCHAR_MAX	Maximum value for signed char
UCHAR_MAX	Maximum value for unsigned char
CHAR_MIN	Minimum value for char
CHAR_MAX	Maximum value for char
MB_LEN_MAX	Maximum number of bytes for a multiple-byte character.
SHRT_MIN	Minimum value for shortint
SHRT_MAX	Maximum value for shortint
USHRT_MAX	Maximum value for unsignedshortint
INT_MIN	Minimum value for int
INT_MAX	Maximum value for int
UINT_MAX	Maximum value for unsignedint
LONG_MIN	Minimum value for longint
LONG_MAX	Maximum value for longint
ULONG_MAX	Maximum value for unsignedlongint

18.4 locale.h

The header file in the following listing defines a struct containing all the locale-specific values.

Listing: Locale-Specific Values

```
struct
lconv {              /*
"
C
"
 locale (default) */
  char *decimal_point;
/* "." */

  /* Decimal point character to use for non-monetary numbers */

  char *thousands_sep;
/* "" */

  /* Character to use to separate digit groups in

     the integral part of a non-monetary number. */
```

```
  char *grouping;
/* "\CHAR_MAX" */

  /* Number of digits that form a group. CHAR_MAX

     means "no grouping",
'\0' means take previous

     value.

     for example, the string
"\3\0" specifies the repeated

     use of groups of three digits. */

  char *int_curr_symbol;
/* "" */

  /* 4-character string for the international

     currency symbol according to ISO 4217. The

     last character is the separator

     between currency symbol and amount. */

  char *currency_symbol;
/* "" */

  /* National currency symbol. */

  char *mon_decimal_point;
/* "." */

  char *mon_thousands_sep;
/* "" */

  char *mon_grouping;
/* "\CHAR_MAX" */

  /* Same as decimal_point etc., but for monetary numbers. */

  char *positive_sign;
/* "" */

  /* String to use for positive monetary numbers.*/

  char *negative_sign;
/* "" */

  /* String to use for negative monetary numbers. */

  char  int_frac_digits;
/* CHAR_MAX */

  /* Number of fractional digits to print in a

     monetary number according to international format. */

  char  frac_digits;
/* CHAR_MAX */

  /* The same for national format. */

  char  p_cs_precedes;
/* 1 */

  /* 1 indicates that the currency symbol is left

     of a positive monetary amount; 0 indicates it is on the right. */
```

```
  char  p_sep_by_space;
/* 1 */

  /* 1 indicates that the currency symbol is

     separated from the number by a space for

     positive monetary amounts. */

  char  n_cs_precedes;
/* 1 */

  char  n_sep_by_space;
/* 1 */

  /* The same for negative monetary amounts. */

  char  p_sign_posn;
/* 4 */

  char  n_sign_posn;
/* 4 */

  /* Defines the position of the sign for positive

     and negative monetary numbers:

     0  amount and currency are in parentheses

     1  sign comes before amount and currency

     2  sign comes after the amount

     3  sign comes immediately before the currency

     4  sign comes immediately after the currency */

};
```

Use one of several constants in setlocale() to define which part of the locale to set. Refer the following table.

Table 18-4. Constants Used with setlocale()

Constant	Description
LC_ALL	Changes the complete locale.
LC_COLLATE	Only changes the locale for functions strcoll() and strxfrm().
LC_MONETARY	Changes the locale for formatting monetary numbers.
LC_NUMERIC	Changes the locale for numeric (non-monetary) formatting.
LC_TIME	Changes the locale for function strftime().
LC_TYPE	Changes the locale for character handling and multiple-byte character functions.

This implementation only supports the minimum C locale.

18.5 math.h

Defines only this constant:

```
HUGE_VAL
```

Large value that is returned if overflow occurs.

18.6 setjmp.h

Contains only this type definition:

```
typedef jmp_buf;
```

A buffer for setjmp() to store the current program state.

18.7 signal.h

Defines signal-handling constants and types. See the following tables.

```
typedef sig_atomic_t;
```

Table 18-5. Constants Defined in signal.h

Constant	Definition
SIG_DFL	If passed as the second argument to signal, the default response is installed.
SIG_ERR	Return value of signal(), if the handler cannot be installed.
SIG_IGN	If passed as the second argument to `signal()`, the signal is ignored.

Table 18-6. Signal-Type Macros

Constant	Definition
SIGABRT	Abort program abnormally
SIGFPE	Floating point error
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGSEGV	Segmentation violation
SIGTERM	Terminate program normally

18.8 stddef.h

Defines a few generally useful types and constants. Refer the following table.

Table 18-7. Constants Defined in stddef.h

Constant	Description
`ptrdiff_t`	The result type of the subtraction of two pointers.
`size_t`	Unsigned type for the result of `sizeof`.
`wchar_t`	Integral type for wide characters.
`#define NULL ((void *) 0)`	
`size_t offsetof ( type, struct_member)`	Returns the offset of field `struct_member` in `struct` type.

18.9 stdio.h

This header file contains two type declarations. Refer the following table.

Table 18-8. Type Definitions in stdio.h

Type Definition	Description
`FILE`	Defines a type for a file descriptor.
`fpos_t`	A type to hold the position in the file as needed by fgetpos() and fsetpos().

The following table lists the constants defined in `stdio.h`.

Table 18-9. Constants Defined in stdio.h

Constant	Description
`BUFSIZ`	Buffer size for setbuf().
`EOF`	Negative constant to indicate end-of-file.
`FILENAME_MAX`	Maximum length of a filename.
`FOPEN_MAX`	Maximum number of open files.
`_IOFBF`	To set full buffering in setvbuf().
`_IOLBF`	To set line buffering in setvbuf().
`_IONBF`	To switch off buffering in setvbuf().
`SEEK_CUR`	fseek() positions relative from current position.

Table continues on the next page...

Table 18-9. Constants Defined in stdio.h (continued)

Constant	Description
SEEK_END	fseek() positions from the end of the file.S
SEEK_SET	fseek() positions from the start of the file.
TMP_MAX	Maximum number of unique filenames tmpnam() can generate.

There are three variables for the standard I/O streams:

```
extern FILE *  stderr,   *stdin,   *stdout;
```

18.10 stdlib.h

Besides a redefinition of NULL, size_t and wchar_t, this header file contains the type definitions listed in the following table.

Table 18-10. Type Definitions in stdlib.h

Type Definition	Description
typedef div_t;	A struct for the return value of div().
typedef ldiv_t;	A struct for the return value of ldiv().

The following table lists the constants defined in stdlib.h.

Table 18-11. Constants Defined in stdlib.h

Constant	Definition
EXIT_FAILURE	Exit code for unsuccessful termination.
EXIT_SUCCESS	Exit code for successful termination.
RAND_MAX	Maximum return value of rand().
MB_LEN_MAX	Maximum number of bytes in a multi-byte character.

18.11 time.h

This header files defines types and constants for time management. Refer the following listing.

Listing: time.h Type Definitions and Constants

```
typedef
clock_t;
typedef
time_t;

struct tm {

  int tm_sec;     /* Seconds */

  int tm_min;     /* Minutes */

  int tm_hour;    /* Hours */

  int tm_mday;    /* Day of month: 0 .. 31 */

  int tm_mon;     /* Month: 0 .. 11 */

  int tm_year;    /* Year since 1900 */

  int tm_wday;    /* Day of week: 0 .. 6 (Sunday == 0) */

  int tm_yday;    /* day of year: 0 .. 365 */

  int tm_isdst;   /* Daylight saving time flag:

                     > 0  It is DST

                       0  It is not DST

                     < 0  unknown */

};
```

The constant `CLOCKS_PER_SEC` gives the number of clock ticks per second.

18.12 string.h

The `string.h` file defines only functions to manipulate ANSI-C string but not types or special defines.

The functions are explained below together with all other ANSI functions.

18.13 assert.h

The `assert.h` file defines the assert() macro. If the `NDEBUG` macro is defined, then assert does nothing. Otherwise, `assert()` calls the auxiliary `function _assert` if the sole macro parameter of `assert()` evaluates to `0 (FALSE)`. Refer the following listing.

Listing: Use assert() to Assist in Debugging

```
#ifdef NDEBUG
  #define assert(EX)

#else

  #define assert(EX) ((EX) ? 0 : _assert(__LINE__, __FILE__))

#endif
```

18.14 stdarg.h

The `stdarg.h` file defines the type `va_list` and the macros va_arg(), va_end(), and va_start(). The type `va_list` implements a pointer to one argument of an open parameter list. The macro `va_start()` initializes a variable of type `va_list` to point to the first open parameter, given the last explicit parameter and its type as arguments. The macro `va_arg()` returns one open parameter, given its type and also makes the `va_list` argument pointing to the next parameter. The `va_end()` macro finally releases the actual pointer. For all implementations, the `va_end()` macro does nothing because `va_list` is implemented as an elementary data type and therefore it must not be released. The `va_start()` and the `va_arg()` macros have a type parameter, accessed only with `sizeof()`.

Listing: Example using stdarg.h

```
char sum(long p, ...) {
  char res=0;

  va_list list= va_start()(p, long);

  res= va_arg(list, int); // (*)

  va_end(list);

  return res;

}

void main(void) {

  char c = 2;

  if (f(10L, c) != 2) Error();

}
```

In the line `(*)`, `va_arg` must be called with `int`, not with `char`. Because of the default argument-promotion rules of C, integral types pass an `int` at least and floating types pass a `double` at least. In other words, using `va_arg(..., char)` or `va_arg(..., short)` yields undefined results in C. Be cautious when using variables instead of types for `va_arg()`. In the example above, `res= va_arg(list, res)` is incorrect unless `res` has type `int` and not `char`.

18.15 ctype.h

The `ctype.h` file defines functions to check properties of characters, as if a character is a digit (`isdigit()`), a space (`isspace()`), or something else. Implement these functions either as macros or as real functions. Use the macro version when using the -Ot compiler option or when the macro `__OPTIMIZE_FOR_TIME__` is defined. The macros use a table called `_ctype`. whose length is 257 bytes. In this array, all properties tested by the various functions are encoded by single bits, taking the character as indices into the array. The function implementations otherwise do not use this table. They save memory by using the shorter call to the function (compared with the expanded macro).

The functions in the following listing are explained below together with all other ANSI functions.

Listing: Macros Defined in ctypes.h

```
extern unsigned char  _ctype[];
#define  _U   (1<<0)     /* Upper case        */

#define  _L   (1<<1)     /* Lower case        */

#define  _N   (1<<2)     /* Numeral (digit)   */

#define  _S   (1<<3)     /* Spacing character */

#define  _P   (1<<4)     /* Punctuation       */

#define  _C   (1<<5)     /* Control character */

#define  _B   (1<<6)     /* Blank             */

#define  _X   (1<<7)     /* hexadecimal digit */

#ifdef __OPTIMIZE_FOR_TIME__ /* -Ot defines this macro */

#define  isalnum(c) (_ctype[(unsigned char)(c+1)] & (_U|_L|_N))

#define  isalpha(c) (_ctype[(unsigned char)(c+1)] & (_U|_L))

#define  iscntrl(c) (_ctype[(unsigned char)(c+1)] & _C)

#define  isdigit(c) (_ctype[(unsigned char)(c+1)] & _N)

#define  isgraph(c) (_ctype[(unsigned char)(c+1)] & (_P|_U|_L|_N))

#define  islower(c) (_ctype[(unsigned char)(c+1)] & _L)

#define  isprint(c) (_ctype[(unsigned char)(c+1)] & (_P|_U|_L|_N|_B))

#define  ispunct(c) (_ctype[(unsigned char)(c+1)] & _P)

#define  isspace(c) (_ctype[(unsigned char)(c+1)] & _S)

#define  isupper(c) (_ctype[(unsigned char)(c+1)] & _U)

#define  isxdigit(c)(_ctype[(unsigned char)(c+1)] & _X)

#define  tolower(c) (isupper(c) ? ((c) - 'A' + 'a') : (c))

#define  toupper(c) (islower(c) ? ((c) - 'a' + 'A') : (c))
```

```
#define  isascii(c) (!((c) & ~127))

#define  toascii(c) (c & 127)

#endif /* __OPTIMIZE_FOR_TIME__ */
```

Chapter 19
The Standard Functions

This section describes all the standard functions in the ANSI-C library. Each function description consists of the subsections listed in the following table:

Table 19-1. Function Description Subsections

Subsection	Description
Syntax	Shows the function's prototype and which header file to include.
Description	Describes how to use the function.
Return	Describes what the function returns in which case. If the function modifies the global variable `errno`, describes possible values also.
See also	Contains cross-references to related functions.

Some of the functions described in this chapter are unimplemented functions. Unimplemented functions are categorized in the following categories:

- Hardware specific
- File I/O

The following listed are the standard functions in the ANSI-C library:

Table 19-2. Standard Functions in ANSI-C library

abort()	abs()	acos() and acosf()
asctime()	asin() and asinf()	assert()
atan() and atanf()	atan2() and atan2f()	atexit()
atof()	atoi()	atol()
bsearch()	calloc()	ceil() and ceilf()
clearerr()	clock()	cos() and cosf()
cosh() and coshf()	ctime()	difftime()
div()	exit()	exp() and expf()
fabs() and fabsf()	fclose()	feof()

Table continues on the next page...

Table 19-2. Standard Functions in ANSI-C library (continued)

ferror()	fflush()	fgetc()
fgetpos()	fgets()	floor() and floorf()
fmod() and fmodf()	fopen()	fprintf()
fputc()	fputs()	fread()
free()	freopen()	frexp() and frexpf()
fscanf()	fseek()	fsetpos()
ftell()	fwrite()	getc()
getchar()	getenv()	gets()
gmtime()	isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit()	labs()
ldexp() and ldexpf()	ldiv()	localeconv()
localtime()	log() and logf()	log10() and log10f()
longjmp()	malloc()	mblen()
mbstowcs()	mbtowc()	memchr()
memcmp()	memcpy() and memmove()	memset()
mktime()	modf() and modff()	perror()
pow() and powf()	printf()	putc()
putchar()	puts()	qsort()
raise()	rand()	realloc()
remove()	rename()	rewind()
scanf()	setbuf()	setjmp()
setlocale()	setvbuf()	signal()
sin() and sinf()	sinh() and sinhf()	sprintf()
sqrt() and sqrtf()	srand()	sscanf()
strcat()	strchr()	strcmp()
strcoll()	strcpy()	strcspn()
strerror()	strftime()	strlen()
strncat()	strncmp()	strncpy()
strpbrk()	strrchr()	strspn()
strstr()	strtod()	strtok()
strtol()	strtoul()	strxfrm()
system()	tan() and tanf()	tanh() and tanhf()
time()	tmpfile()	tmpnam()
tolower()	toupper()	ungetc()
va_arg(), va_end(), and va_start()	vfprintf(), vprintf(), and vsprintf()	wctomb()
wcstombs()		

19.1 abort()

Syntax

```
#include <stdlib.h >
```

void abort(void);

Description

`abort()` terminates the program. It does the following (in this order):

- Raises signal `SIGABRT`
- Flushes all open output streams
- Closes all open files
- Removes all temporary files
- Calls `HALT`

If your application handles `SIGABRT` and the signal handler does not return (for example, because it does a longjmp()), the application is not halted.

Return

None

See also

`atexit()`

`exit()`

`raise()`

`signal()`

19.2 abs()

Syntax

```
#include <stdlib.h >
```

int abs(int i);

Description

`abs()` computes the absolute value of `i`.

Return

The absolute value of `i`; that is, `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-32,768`, this value is returned and `errno` is set to `ERANGE`.

See also

`fabs() and fabsf()`

`labs()`

19.3 acos() and acosf()

Syntax

```
#include <math.h >

double acos (double x);

float  acosf (float x);
```

Description

`acos()` computes the principal value of the arc cosine of `x`.

Return

The arc cosine `cos^(-1)(x)` of `x` in the range between `0` and Pi, if `x` is in the range `-1 <= x <= 1`. If `x` is not in this range, the function returns `NAN` and sets `errno` to `EDOM`.

See also

asin() and asinf()

atan() and atanf()

atan2() and atan2f()

19.4 asctime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


char * asctime(const struct tm* timeptr);
```

Description

`asctime()` converts the time, broken down in `timeptr`, into a string.

Return

A pointer to a string containing the time string.

See also

`localtime()`

`mktime()`

`time()`

19.5 asin() and asinf()

Syntax

```
#include <math.h >
```

```
double asin(double x);


float  asinf(float x);
```

Description

`asin()` computes the principal value of the arc sine of `x`.

Return

The arc sine `sin^(-1)(x)` of `x` in the range between `-Pi/2` and `Pi/2`, if `x` is in the range `-1 <= x <= 1`. If `x` is not in this range, the function returns `NAN` and sets `errno` to `EDOM`.

See also

acos() and acosf()

atan() and atanf()

atan2() and atan2f()

`cos() and cosf()`

tan() and tanf()

19.6 assert()

Syntax

```
#include <assert.h >


void assert(int expr);
```

Description

The `assert()` macro indicates that expression `expr` is expected to be true at this point in the program. If `expr` is false (0), `assert()` halts the program. Compiling with option `-DNDEBUG` or placing the preprocessor control statement `#define NDEBUG` before the `#include <assert.h>` statement effectively deletes all assertions from the program.

Return

Nothing

See also

abort()

exit()

19.7 atan() and atanf()

Syntax

```
#include <math.h >


double atan (double x);


float  atanf(float x);
```

Description

`atan()` computes the principal value of the arc tangent of `x`.

Return

The arc tangent `tan^(-1)(x)`, in the range from `-Pi/2` to `Pi/2` rad.

See also

acos() and acosf(),

asin() and asinf(),

atan2() and atan2f(),

cos() and cosf(),

sin() and sinf(), and

tan() and tanf()

19.8 atan2() and atan2f()

Syntax

```
#include <math.h >


double atan2(double y, double x);


float  atan2f(float y, float x);
```

Description

`atan2()` computes the principal value of the arc tangent of `y/x`. It uses the sign of both operands to determine the quadrant of the result.

Return

The arc tangent `tan^(-1)(y/x)`, in the range from `-Pi` to `Pi` rad, if neither `x` or `y` are `0`. If both `x` and `y` are `0`, it returns `0`.

See also

`acos() and acosf()`

`asin() and asinf()`

`atan() and atanf()`

`cos() and cosf()`

`sin() and sinf()`

`tan() and tanf()`

19.9 atexit()

Syntax

```
#include <stdlib.h >
```

```
int atexit(void (*func) (void));
```

Description

`atexit()` lets you install a function that is to be executed just before the normal termination of the program. You can register at most 32 functions with `atexit()`. The first function registered is the last function called; likewise the last function registered is the first function called.

Return

`atexit()` returns 0 if it registered the function, otherwise it returns a non-zero value.

See also

`abort()`

`exit()`

19.10 atof()

Syntax

```
#include <stdlib.h >


double atof(const char *s);
```

Description

`atof()` converts the string `s` to a `double` floating point value, ignoring white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atof()` is as follows:

```
FloatNum        = Sign{Digit}[.{Digit}][Exp]
```

```
Sign            = [+|-]

Digit           = <any decimal digit from 0 to 9>

Exp             = (e|E) SignDigit{Digit}
```

Return

`atof()` returns the converted `double` floating point value.

See also

atoi()

`strtod()`

strtol()

strtoul()

19.11 atoi()

Syntax

```
#include <stdlib.h >

int atoi(const char *s);
```

Description

`atoi()` converts the string `s` to an integer value, ignoring white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atoi()` is as follows:

```
Number              = [+|-]Digit{Digit}
```

Return

`atoi()` returns the converted integer value.

See also

atof()

atol()

strtod()

strtol()

strtoul()

19.12 atol()

Syntax

```
#include <stdlib.h >



long atol(const char *s);
```

Description

`atol()` converts the string `s` to an `long` value, ignoring white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by the followingatol() is as follows:

```
Number = [+|-]Digit{Digit}
```

Return

`atol()` returns the converted `long` value.

See also

atoi()

atof()

strtod()

strtol()

strtoul()

19.13 bsearch()

Syntax

```
#include <stdlib.h >


void *bsearch(const void *key,

              const void *array,

              size_t n,

              size_t size,

              cmp_func cmp());
```

Description

`bsearch()` performs a binary search in a sorted array. It calls the comparison function `cmp()` with two arguments: a pointer to the key element that is to be found and a pointer to an array element. Thus, the `cmp_func` type can be declared as:

```
typedef int (*cmp_func)(const void *key,

 const void
*data);
```

The comparison function returns an integer according to the following table.

Table 19-3. cmp_func() Return Value

Key Element	Expected Return Value
Less than the array element	Less than zero (negative)
Equal to the array element	Zero
Greater than the array element	Greater than zero (positive)

The following table lists the arguments of bsearch() are.

Table 19-4. Possible bsearch() Function Arguments

Parameter Name	Meaning
`key`	A pointer to the key data you are seeking
`array`	A pointer to the beginning (that is, the first element) of the array that is searched
`n`	The number of elements in the array
`size`	The size (in bytes) of one element in the table
`cmp()`	The comparison function

NOTE

Make sure the array contains only elements of the same size. `bsearch()` also assumes that the array is sorted in ascending order with respect to the comparison function `cmp()`.

Return

`bsearch()` returns a pointer to an array element that matches the key, if one exists. If the comparison function never returns zero (that is, no matching array element exists), `bsearch()` returns `NULL`.

19.14 calloc()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >
```

```
void *calloc(size_t n, size_t size);
```

Description

`calloc()` allocates a block of memory for an array containing `n` elements of size `size`, and initializes all bytes in the memory block to zero. To deallocate the block, use free(). Do not use the default implementation in interrupt routines as it is not reentrant.

Return

`calloc()` returns a pointer to the allocated memory block. If `calloc()` cannot allocate the block, the return value is `NULL`.

See also

`malloc()`

`realloc()`

19.15 ceil() and ceilf()

Syntax

```
#include <math.h >


double ceil(double x);


float  ceilf(float x);
```

Description

These functions round their parameters up to the nearest integer.

Return

`ceil()` returns the smallest integral number larger than `x`.

See also

floor() and floorf()

fmod() and fmodf()

19.16 clearerr()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >

void clearerr(FILE *f);
```

Description

`clearerr()` resets the error flag and the `EOF` marker of file `f`.

Return

None

19.17 clock()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >

clock_t clock(void);
```

Description

`clock()` determines the amount of time since your system started, in clock ticks. To convert to seconds, divide by `CLOCKS_PER_SEC`.

Return

`clock()` returns the amount of time since system startup.

See also

time()

19.18 cos() and cosf()

Syntax

```
#include <time.h >


double cos (double x);


float  cosf(float x);
```

Description

`cos()` computes the principal value of the cosine of `x`. Express `x` in radians.

Return

The cosine `cos(x)`.

See also

acos() and acosf()

asin() and asinf()

atan() and atanf()

atan2() and atan2f()

sin() and sinf()

tan() and tanf()

19.19 cosh() and coshf()

Syntax

```
#include <time.h >


double cosh (double x);


float  coshf(float x);
```

Description

`cosh()` computes the hyperbolic cosine of `x`.

Return

The hyperbolic cosine `cosh(x)`. If the computation fails because the value is too large, the function returns `HUGE_VAL` and sets `errno` to `ERANGE`.

See also

cos() and cosf()

sinh() and sinhf()

tanh() and tanhf()

19.20 ctime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


char *ctime(const time_t *timer);
```

Description

`ctime()` converts the calendar-time timer to a character string.

Return

The string containing the ASCII representation of the date.

Seealso

asctime()

mktime()

time()

19.21 difftime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


double difftime(time_t *t1, time_t t0);
```

Description

`difftime()` calculates the number of seconds between any two calendar times.

Return

The number of seconds between the two times, as a `double`.

See also

mktime()

time()

19.22 div()

Syntax

```
#include <stdlib.h >
```

```
div_t div(int x, int y);
```

Description

`div()` computes both the quotient and the modulus of the division `x/y`.

Return

A structure with the results of the division.

See also

`ldiv()`

19.23 exit()

Syntax

```
#include <stdlib.h >


void exit(int status);
```

Description

`exit()` terminates the program normally. It does the following, in this order:

- executes all functions registered with atexit()
- flushes all open output streams
- closes all open files
- removes all temporary files
- calls `HALT`

`exit()` ignores the `status` argument.

See also

abort()

19.24 exp() and expf()

Syntax

```
#include <math.h >


double exp (double x);


float  expf(float x);
```

Description

`exp()` computes `ex`, where `e` is the base of natural logarithms.

Return

`ex`. If the computation fails because the value is too large, this function returns `HUGE_VAL` and sets `errno` to `ERANGE`.

See also

`log() and logf()`

`log10() and log10f()`

`pow() and powf()`

19.25 fabs() and fabsf()

Syntax

```
#include <math.h >


double fabs (double x);


float  fabsf(float x);
```

Description

`fabs()` computes the absolute value of `x`.

Return

The absolute value of `x` for any value of `x`.

See also

abs()

labs()

19.26 fclose()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


int fclose(FILE *f);
```

Description

`fclose()` closes file f. Before doing so, `fclose()` does the following:

- Flushes the stream, if the file was not opened in read-only mode
- Discards and deallocates any buffers that were allocated automatically (that is, not using setbuf()).

Return

Zero, if the function succeeds; `EOF` otherwise.

See also

fopen()

19.27 feof()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int feof(FILE *f);
```

Description

`feof()` tests whether previous I/O calls on file `f` tried to do anything beyond the end of the file.

NOTE

Calling `clearerr()` or fseek() clears the file's `end-of-file` flag; therefore `feof()` returns 0.

Return

Zero, if not at the end of the file; `EOF` otherwise.

19.28 ferror()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int ferror(FILE *f);
```

Description

`ferror()` tests whether an error occurred on file `f`. To clear the file's error indicator, use clearerr(). rewind() automatically resets the file's error flag.

NOTE

Do not use `ferror()` to test for `end-of-file`. Use feof() instead.

Return

Zero, if no error occurred; non-zero otherwise.

19.29 fflush()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fflush(FILE *f);
```

Description

`fflush()` flushes the I/O buffer of file `f`, allowing a clean switch between reading and writing the same file. If the program was writing to file `f`, `fflush()` writes all buffered data to the file. If it was reading, `fflush()` discards any buffered data. If `f` is `NULL`, `fflush()` flushes *all* files open for writing.

Return

Zero, if no error occurred; `EOF` otherwise.

See also

setbuf()

setvbuf()

19.30 fgetc()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fgetc(FILE *f);
```

Description

`fgetc()` reads the next character from file `f`.

NOTE

If file `f` was opened as a text file, `fgetc()` reads the end-of-line character combination as one `'\n'` character.

Return

`fgetc()` reads the character as an integer in the range from 0 to 255. If a read error occurs, `fgetc()` returns `EOF` and sets the file's error flag, so that a subsequent call to ferror() returns a non-zero value. If an attempt is made to read beyond the end of the file, `fgetc()` also returns `EOF`, `but` sets the end-of-file flag instead of the error flag so that feof() returns `EOF`, but `ferror()` returns zero.

See also

fgets()

fopen()

fread()

fscanf()

getc()

19.31 fgetpos()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >
```

```
int fgetpos(FILE *f, fpos_t *pos);
```

Description

`fgetpos()` returns the current file position in `*pos`. Use this value later to reset the position, using fsetpos().

NOTE

Do *not* assume the value in `*pos` to have any particular meaning, such as a byte offset from the beginning of the file. The ANSI standard does not require this, and in fact any value may be put into `*pos` as long as an `fsetpos()` with that value resets the position in the file correctly.

Return

Non-zero, if an error occurred; zero otherwise.

See also

fseek()

ftell()

19.32 fgets()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


char *fgets(char *s, int n, FILE *f);
```

Description

`fgets()` reads a string of at most `n-1` characters from file `f` into `s`. Immediately after the last character read, `fgets()` appends a `'\0'`. If `fgets()` reads a line break (`'\n'`) or reaches the end of the file before having read `n-1` characters, the following happens:

- If `fgets()` reads a line break, it adds the `'\n'` plus a `'\0'` to `s` and returns successfully.
- If it reaches the end of the file after having read at least 1 character, it adds a `'\0'` to s and returns successfully.
- If it reaches `EOF` without having read any character, it sets the file's end-of-file flag and returns unsuccessfully (`s` is left unchanged).

Return

`NULL`, if an error occurred; `s` otherwise.

See also

fgetc()

fputs()

19.33 floor() and floorf()

Syntax

```
#include <math.h >


double floor (double x);


float  floorf(float x);
```

Description

`floor()` calculates the largest integral number less than or equal to `x`.

Return

The largest integral number not less than or equal to `x`.

See also

ceil() and ceilf()

modf() and modff()

19.34 fmod() and fmodf()

Syntax

```
#include <math.h >


double fmod (double x, double y);


float  fmodf(float x, float y);
```

Description

`fmod()` calculates the floating point remainder of `x/y`.

Return

The floating point remainder of `x/y`, with the same sign as `x`. If `y` is `0`, it returns `0` and sets `errno` to `EDOM`.

Seealso

div()

ldiv()

ldexp() and ldexpf()

modf() and modff()

19.35 fopen()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >
```

```
FILE *fopen(const char *name, const char *mode);
```

Description

`fopen()` opens a file with the given name and mode. It automatically allocates an I/O buffer for the file.

There are three main modes: read, write, and update (both read and write) accesses. Each can be combined with either text or binary mode to read a text file or update a binary file. Opening a file for text accesses translates the end-of-line character (combination) into `'\n'` when reading and vice versa when writing. The following table lists all possible modes.

Table 19-5. Operating Modes of the fopen() Function

Mode	Effect
r	Open the file as a text file for reading.
w	Create a text file and open it for writing.
a	Open the file as a text file for appending.
rb	Open the file as a binary file for reading.
wb	Create a file and open as a binary file for writing.
ab	Open the file as a binary file for appending.
r+	Open a text file for updating.
w+	Create a text file and open for updating.
a+	Open a text file for updating. Append all writes to the end.
r+b or rb+	Open a binary file for updating.
w+b or wb+	Create a binary file and open for updating.
a+b or ab+	Open a binary file for updating. Append all writes to the end.

If the mode contains an `r`, but the file doesn't exist, `fopen()` returns unsuccessfully. Opening a file for appending (mode contains `a`) always appends writing to the end, even if fseek(), fsetpos(), or rewind() is called. Opening a file for updating allows both read and write accesses on the file. However, fseek(), fsetpos(), or rewind() must be called in order to write after a read or to read after a write.

Return

A pointer to the file descriptor of the file. If `fopen()` cannot create the file, the function returns `NULL`.

See also

fclose()

freopen()

setbuf()

setvbuf()

19.36 fprintf()

Syntax

```
#include <stdio.h >


int fprintf(FILE *f, const char *format, ...);
```

Description

`fprintf()` is the same as sprintf(), but the output goes to file `f` instead of a string.

For a detailed format description, see `sprintf()`.

Return

The number of characters written. If some error occurred, `fprintf()` returns `EOF`.

See also

printf()

vfprintf(), vprintf(), and vsprintf()

19.37 fputc()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fputc(int ch, FILE *f);
```

Description

`fputc()` writes a character to file `f`.

Return

The integer value of `ch`. If an error occurred, `fputc()` returns `EOF`.

See also

fputs()

19.38 fputs()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fputs(const char *s, FILE *f);
```

Description

`fputs()` writes the zero-terminated string `s` to file `f` (without the terminating `'\0'`).

Return

`EOF`, if an error occurred; zero otherwise.

See also

fputc()

19.39 fread()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


size_t fread(void *ptr, size_t size, size_t n, FILE *f);
```

Description

`fread()` reads a contiguous block of data. It attempts to read `n` items of size `size` from file `f` and stores them in the array to which `ptr` points. If either `n` or `size` is `0`, nothing is read from the file and the array is left unchanged.

Return

The number of items successfully read.

See also

fgetc()

fgets()

fwrite()

19.40 free()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


void free(void *ptr);
```

Description

`free()` deallocates a memory block previously allocated by calloc(), malloc(), or realloc(). If `ptr` is `NULL`, nothing happens. The default implementation is not reentrant; do not use in interrupt routines.

Return

19.41 freopen()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


void freopen(const char *name,

             const char *mode,

             FILE *f);
```

Description

`freopen()` opens a file using a specific file descriptor. Use this function for redirecting `stdin`, `stdout`, or `stderr`. For information about possible modes, see fopen().

Return

See also

`fclose()`

fopen()

19.42 frexp() and frexpf()

Syntax

```
#include <math.h >
```

```
double frexp(double x, int *exp);


float  frexpf(float x, int *exp);
```

Description

`frexp()` splits a floating point number into mantissa and exponent. The relation is `x = m * 2^exp`. `m` always normalizes to the range `0.5 < m <= 1.0`. The mantissa has the same sign as `x`.

Return

The mantissa of `x` (the exponent is written to `*exp`). If `x` is `0.0`, both the mantissa (the return value) and the exponent are `0`.

See also

exp() and expf()

ldexp() and ldexpf()

modf() and modff()

19.43 fscanf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fscanf(FILE *f, const char *format, ...);
```

Description

`fscanf()` is the same as scanf() but the input comes from file `f` instead of a string.

Return

The number of data arguments read, if any input was converted. If not, it returns `EOF`.

See also

fgetc()

fgets()

sscanf()

19.44 fseek()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fseek(FILE *f, long offset, int mode);
```

Description

`fseek()` sets the current position in file `f`.

For binary files, the position can be set in three ways, as shown in this table.

Table 19-6. fseek() Function Offset Position into the File

Mode	Set Position
SEEK_SET	`offset` bytes from the beginning of the file.
SEEK_CUR	`offset` bytes from the current position.
SEEK_END	`offset` bytes from the end of the file.

For text files, either `offset` must be zero or `mode` is `SEEK_SET` and `offset` a value returned by a previous call to ftell().

If `fseek()` is successful, it clears the file's end-of -file flag. The position cannot be set beyond the end of the file.

Return

Zero, if successful; non-zero otherwise.

See also

fgetpos()

fsetpos()

ftell()

19.45 fsetpos()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int fsetpos(FILE *f, const fpos_t *pos);
```

Description

`fsetpos()` sets the file position to `pos`, which must be a value returned by a previous call to fgetpos() on the same file. If the function is successful, it clears the file's end-of-file flag.

The position cannot be set beyond the end of the file.

Return

Zero, if successful; non-zero otherwise.

See also

fgetpos()

fseek()

ftell()

19.46 ftell()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


long ftell(FILE *f);
```

Description

`ftell()` returns the current file position. For binary files, this is the byte offset from the beginning of the file; for text files, do not use this value except as an argument to `fseek()`.

Return

-1, if an error occurred; otherwise the current file position.

See also

fgetpos()

fsetpos()

19.47 fwrite()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


size_t fwrite(const void *p,

              size_t size,

              size_t n,
```

```
                FILE *f);
```

Description

`fwrite()` writes a block of data to file f. It writes `n` items of size `size`, starting at address `ptr`.

Return

The number of items successfully written.

See also

fputc()

fputs()

fread()

19.48 getc()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int getc(FILE *f);
```

Description

`getc()` is the same as `fgetc()`, but may be implemented as a macro. Therefore, ensure that expression `f` has no side effects.

Return

See also

`fgetc()`

19.49 getchar()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >

int getchar(void);
```

Description

`getchar()` is the same as `getc(stdin)`.

Return

See also

fgetc()

19.50 getenv()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >

char *getenv(const char *name);
```

Description

`getenv()` returns the value of environment variable `name`.

Return

NULL

See also

19.51 gets()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


char *gets(char *s);
```

Description

gets() reads a string from stdin and stores it in s. It stops reading when it reaches a line break or EOF character, but does not append this character to the string. gets() terminates the string with zero.

If the function reads EOF before any other character, it sets stdin's end-of-file flag and returns unsuccessfully without changing string s.

Return

NULL, if an error occurred; s otherwise.

See also

fgetc()

puts()

19.52 gmtime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


struct tm *gmtime(const time_t *time);
```

Description

`gmtime()` converts `*time` to Universal Coordinated Time (UTC), which is equivalent to Greenwich Mean Time (GMT).

Return

`NULL`, if UTC is unavailable; a pointer to a `struct` containing UTC otherwise.

See also

`ctime()`

`time()`

19.53 isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit()

Syntax

```
#include <ctype.h >


int isalnum (int ch);


int isalpha (int ch);


...


int isxdigit(int ch);
```

Description

These functions determine whether character `ch` belongs to a certain set of characters. The next table describes the character ranges tested by the functions.

Table 19-7. Testing Functions Character Range

Function	ch Test Range
`isalnum()`	Alphanumeric character (`A-Z`, `a-z`, or `0-9`)
`isalpha()`	Alphabetic character (`A-Z` or `a-z`)
`iscntrl()`	Control character (`\000-\037` or `\177 (DEL)`)
`isdigit()`	Decimal digit (`0-9`)
`isgraph()`	Printable character except space (`!-~`)
`islower()`	Lower case letter (`a-z`)
`isprint()`	Printable character (`' '-~`)
`ispunct()`	Punctuation character (`!-/`, `:-@`, `[-'` and `{-~`)
`isspace()`	White space character (`' '`, `\f`, `\n`, `\r`, `\t` and `\v`)
`isupper()`	Upper case letter (`A-Z`)
`isxdigit()`	Hexadecimal digit (`0-9`, `A-F` or `a-f`)

Return

TRUE (`1`), if `ch` is in the character class; `0` otherwise.

See also

tolower()

toupper()

19.54 labs()

Syntax

```
#include <stdlib.h >


long labs(long i);
```

Description

`labs()` computes the absolute value of `i`.

Return

The absolute value of `i` (`i` if `i` is positive and `-i` if `i` is negative). If `i` is `-2,147,483,648`, `labs()` returns this value and sets `errno` to `ERANGE`.

See also

abs()

19.55 ldexp() and ldexpf()

Syntax

```
#include <math.h >


double ldexp (double x, int exp);


float  ldexpf(float x, int exp);
```

Description

`ldexp()` multiplies `x` by `2exp`.

Return

`x * 2exp`. If this function fails because the result is too large, both `ldexp()` and `ledxpf()` return `HUGE_VAL` and set `errno` to `ERANGE`.

See also

exp() and expf()

frexp() and frexpf()

log() and logf()

log10() and log10f()

modf() and modff()

19.56 ldiv()

Syntax

```
#include <stdlib.h >


ldiv_t ldiv(long x, long y);
```

Description

`ldiv()` computes both the quotient and the modulus of the division `x/y`.

Return

A structure containing the division results.

See also

div()

19.57 localeconv()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <locale.h >


struct lconv *localeconv(void);
```

Description

`localeconv()` returns a pointer to a `struct` containing information about the current locale (for example, formatting monetary quantities).

Return

A pointer to a `struct` containing the desired information.

See also

setlocale()

19.58 localtime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


struct tm *localetime(const time_t *time);
```

Description

`localtime()` converts `*time` into broken-down time.

Return

A pointer to a `struct` containing the broken-down time.

See also

asctime()

mktime()

time()

19.59 log() and logf()

Syntax

```
#include <math.h >
```

```
double log (double x);


float  logf(float x);
```

Description

`log()` computes the natural logarithm of `x`.

Return

`ln(x)`, if `x` is greater than zero. If `x` is smaller then zero, `log()` returns `NAN`; if `x` is equal to zero, `log()` returns negative infinity. In both cases, `log()` sets `errno` to `EDOM`.

See also

exp() and expf()

log10() and log10f()

19.60 log10() and log10f()

Syntax

```
#include <math.h >


double log10(double x);


float  log10f(float x);
```

Description

`log10()` computes the decadic logarithm (the logarithm to base 10) of `x`.

Return

`log10(x)`, if `x` is greater than zero. If `x` is smaller then zero, `log10()` returns `NAN`; if `x` is equal to zero, `log10()` returns negative infinity. In both cases, `log10()` sets `errno` to `EDOM`.

See also

exp() and expf()

log10() and log10f()

19.61 longjmp()

Syntax

```
#include <setjmp.h >


void longjmp(jmp_buf env, int val);
```

Description

`longjmp()` performs a non-local jump to some location, earlier in the call chain, which was marked by a call to setjmp(). `longjmp()` restores the environment at the time of that call to `setjmp()` (`env`, which also was the parameter to setjmp()) and your application continues as if the call to `setjmp()` just returned the value `val`.

Return

See also

`setjmp()`

19.62 malloc()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >
```

```
void *malloc(size_t size);
```

Description

malloc() allocates a block of memory for an object of size size bytes. The content of this memory block is undefined. To deallocate the block, use free(). Do not use malloc() in interrupt routines as the default implementation is not reentrant.

Return

malloc() returns a pointer to the allocated memory block. If the block cannot be allocated, the return value is NULL.

See also

calloc()

realloc()

19.63 mblen()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


int mblen(const char *s, size_t n);
```

Description

mblen() determines the number of bytes the multi-byte character pointed to by s occupies.

Return

0, if s is NULL.

-1, if the first n bytes of *s do not form a valid multi-byte character.

n, the number of bytes of the multi-byte character otherwise.

See also

mbtowc()

mbstowcs()

19.64 mbstowcs()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


size_t mbstowcs(wchar_t *wcs,

                const char *mbs,

                size_t n);
```

Description

`mbstowcs()` converts a multi-byte character string `mbs` to a wide character string `wcs`. Only the first `n` elements are converted.

Return

The number of elements converted, or `(size_t) - 1` if an error occurred.

See also

mblen()

mbtowc()

19.65 mbtowc()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


int mbtowc(wchar_t *wc, const char *s, size_t n);
```

Description

`mbtowc()` converts a multi-byte character `s` to a wide character code `wc`. Only the first `n` bytes of `*s` are taken into consideration.

Return

The number of bytes of the multi-byte character converted `(size_t) if successful or -1 if an error occurred`.

See also

mblen()

mbstowcs()

19.66 memchr()

Syntax

```
#include <string.h >


void *memchr(const void *p, int ch, size_t n);
```

Description

`memchr()` looks for the first occurrence of a byte containing (`ch&0xFF`) in the first *n* bytes of the memory are pointed to by `p`.

Return

A pointer to the byte found, or NULL if no such byte was found.

See also

memcmp()

strchr()

strrchr()

19.67 memcmp()

Syntax

```
#include <string.h >

void *memcmp(const void *p,

             const void *q,

             size_t n);
```

Description

memcmp() compares the first n bytes of the two memory areas pointed to by p and q.

Return

A positive integer, if p is considered greater than q; a negative integer if p is considered smaller than q or zero if the two memory areas are equal.

See also

memchr()

strcmp()

strncmp()

19.68 memcpy() and memmove()

Syntax

```
#include <string.h >

void *memcpy(const void *p,
             const void *q,
             size_t n);

void *memmove(const void *p,
              const void *q,
              size_t n);
```

Description

Both functions copy `n` bytes from `q` to `p`. `memmove()` also works if the two memory areas overlap.

Return

`p`

See also

strcpy()

strncpy()

19.69 memset()

Syntax

```
#include <string.h >


void *memset(void *p, int val, size_t n);
```

Description

`memset()` sets the first `n` bytes of the memory area pointed to by `p` to the value (`val&0xFF`).

Return

`p`

See also

calloc()

`memcpy() and memmove()`

19.70 mktime()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <string.h >


time_t mktime(struct tm *time);
```

Description

`mktime()` converts `*time` to a `time_t`. The `*time` fields may have any value and are not restricted to the ranges given in `time.h`. If the conversion is successful, `mktime()` restricts the fields of `*time` to these ranges and also sets the `tm_wday` and `tm_yday` fields correctly.

Return

`*time` as a `time_t`

See also

ctime()

gmtime()

time()

19.71 modf() and modff()

Syntax

```
#include <math.h >


double modf(double x, double *i);


float  modff(float x, float *i);
```

Description

`modf()` splits the floating-point number `x` into an integral part (returned in `*i`) and a fractional part. Both parts have the same sign as `x`.

Return

The fractional part of `x`

See also

floor() and floorf()

fmod() and fmodf()

frexp() and frexpf()

ldexp() and ldexpf()

19.72 perror()

Syntax

```
#include <stdio.h >


void perror(const char *msg);
```

Description

`perror()` writes an error message appropriate for the current value of `errno` to `stderr`. The character string `msg` is part of `perror`'s output.

Return

See also

assert()

strerror()

19.73 pow() and powf()

Syntax

```
#include <math.h >


double pow (double x, double y);


float  powf(float x, float y);
```

Description

`pow()` computes `x` to the power of `y` (that is, `xy`).

Return

`xy`, if `x > 0`

`1`, if `y == 0`

+x, if (x == 0 && y < 0)

NAN, if (x < 0 && y is not integral). Also, sets errno to EDOM.

±x, with the same sign as x, if the result is too large.

See also

exp() and expf()

ldexp() and ldexpf()

log() and logf()

modf() and modff()

19.74 printf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int printf(const char *format, ...);
```

Description

printf() is the same as sprintf(), but the output goes to stdout instead of a string.

For a detailed format description see sprintf().

Return

The number of characters written. Returns EOF if some error occurred.

See also

fprintf()

vfprintf(), vprintf(), and vsprintf()

19.75 putc()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int putc(char ch, FILE *f);
```

Description

`putc()` is the same as `fputc()`, but may be implemented as a macro. Therefore, ensure that expression `f` has no side effects. See fputc() for more information.

Return

See also

fputc()

19.76 putchar()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int putchar(char ch);
```

Description

`putchar(ch)` is the same as putc `(ch, stdin)`. See fputc() for more information.

Return

See also

fputc()

19.77 puts()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >

int puts(const char *s);
```

Description

`puts()` writes string `s` followed by a newline (`\n`) to `stdout`.

Return

`EOF`, if an error occurred; zero otherwise.

See also

fputc()

putc()

19.78 qsort()

Syntax

```
#include <stdlib.h >

void *qsort(const void *array, size_t n,

            size_t size, cmp_func cmp);
```

Description

`qsort()` sorts the array according to the ordering implemented by the comparison function. It calls the `cmp()` comparison function with two pointers to array elements (`*key`, `*other`). Thus, the type `cmp_func()` can be declared as:

```
typedef int (*cmp_func)(const void *key,

 const void *other);
```

The comparison function returns an integer according to the following table.

Table 19-8. cmp_func() Return Value

`*key` Element Value	Return Value
Less than `*other`	Less than zero (negative)
Equal to `*other`	Zero
Greater than `*other`	Greater than zero (positive)

The arguments to `qsort()` are listed in the following table.

Table 19-9. Possible Arguments for qsort() Function

Argument Name	Meaning
`array`	A pointer to the beginning (the first element) of the array to be sorted
`n`	The number of elements in the array
`size`	The size (in bytes) of one element in the table
`cmp()`	The comparison function

NOTE

Make sure the array contains elements of the same size.

Return

See also

19.79 raise()

Syntax

```
#include <signal.h >


int raise(int sig);
```

Description

`raise()` raises the given signal, invoking the signal handler or performing the defined response to the signal. If a response was undefined or a signal handler is not installed, the application aborts.

Return

Non-zero, if an error occurred; zero otherwise.

See also

signal()

19.80 rand()

Syntax

```
#include <stdlib.h >


int rand(void);
```

Description

`rand()` generates a pseudo random number in the range from `0` to `RAND_MAX`. The numbers generated are based on a seed, which initially is `1`. To change the seed, use srand().

The same seeds always lead to the same sequence of pseudo random numbers.

Return

A pseudo random integer in the range from `0` to `RAND_MAX`.

See also

19.81 realloc()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


void *realloc(void *ptr, size_t size);
```

Description

`realloc()` changes the size of a memory block, preserving its contents. `ptr` must be a pointer returned by calloc(), malloc(), realloc(), or `NULL`. In the latter case, realloc() is equivalent to malloc().

If the new memory block size is smaller than the old size, `realloc()` discards the memory at the end of the block. If size is zero (and `ptr` is not `NULL`), `realloc()` frees the whole memory block.

If there is not enough memory to perform the `realloc()`, the old memory block is left unchanged, and `realloc()` returns `NULL`. Do not use `realloc()` in interrupt routines as the default implementation is not reentrant.

Return

`realloc()` returns a pointer to the new memory block. If the operation cannot be performed, `realloc()` returns `NULL`.

See also

free()

19.82 remove()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h>


  int remove(const char *filename);
```

Description

`remove()` deletes the file `filename`. If the file is open, `remove()` does not delete it and returns unsuccessfully.

Return

Non-zero, if an error occurred; zero otherwise.

See also

tmpfile()

tmpnam()

19.83 rename()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h>


  int rename(const char *from, const char *to);
```

Description

`rename()` renames the `from` file to `to`. If there already is a `to` file, `rename()` does not change anything and returns with an error code.

Return

Non-zero, if an error occurred; zero otherwise.

See also

tmpfile()

tmpnam()

19.84 rewind()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h>


  void rewind(FILE *f);
```

Description

`rewind()` resets the current position in file `f` to the beginning of the file. It also clears the file's error indicator.

Return

See also

fopen()

fseek()

fsetpos()

19.85 scanf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
  #include <stdio.h >
```

```
int scanf(const char *format, ...);
```

Description

`scanf()` is the same as sscanf(), but the input comes from `stdin` instead of a string.

Return

The number of data arguments read, if any input was converted. If not, it returns `EOF`.

See also

fgetc()

fgets()

fscanf()

19.86 setbuf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h>


  void setbuf(FILE *f, char *buf);
```

Description

`setbuf()` lets you specify how a file is buffered. If `buf` is `NULL`, the file is unbuffered (all input or output goes directly to and comes directly from the file). If `buf` is not `NULL`, it is used as a buffer (`buf` points to an array of `BUFSIZ` bytes).

Return

See also

fflush()

setvbuf()

19.87 setjmp()

Syntax

```
#include <setjmp.h >


int setjmp(jmp_buf env);
```

Description

`setjmp()` saves the current program state in the environment buffer `env` and returns zero. This buffer can be used as a parameter to a later call to longjmp(), which then restores the program state and jumps back to the location of `setjmp()`. This time, `setjmp()` returns a non-zero value, which is equal to the second parameter to `longjmp()`.

Return

Zero if called directly. Non-zero if called by longjmp().

See also

longjmp()

19.88 setlocale()

This function is hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <locale.h >


char *setlocale(int class, const char *loc);
```

Description

`setlocale()` changes the program's locale, either all or part of it, depending on `class`. The character string `loc` gives the new locale. The following table lists the classes allowed.

Table 19-10. setlocale() Function Allowable Classes

Class	Affected Function
LC_ALL	All classes
LC_COLLATE	strcoll() and strxfrm() functions
LC_MONETARY	Monetary formatting
LC_NUMERIC	Numeric formatting
LC_TIME	strftime() function
LC_TYPE	Character handling and multi-byte character functions

NOTE

The CodeWarrior IDE supports only the minimum locale C (see locale.h), so this function has no effect.

Return

C, if `loc` is C or `NULL`; `NULL` otherwise.

See also

localeconv()

strcoll()

strftime()

strxfrm()

19.89 setvbuf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h>


  void setvbuf(FILE *f,
```

```
                char *buf,


                int mode,


                size_t size);
```

Description

Use `setvbuf()` to specify how a file is buffered. `mode` determines how the file is buffered.

Table 19-11. setvbuf() Function Operating Modes

Mode	Buffering
_IOFBF	Fully buffered
_IOLBF	Line buffered
_IONBF	Unbuffered

To make a file unbuffered, call `setvbuf()` with mode `_IONBF`; the other arguments (`buf` and `size`) are ignored.

In all other modes, the file uses buffer `buf` of size `size`. If `buf` is `NULL`, the function allocates a buffer of size `size` itself.

See also

fflush()

setbuf()

19.90 signal()

Syntax

```
#include <signal.h >


_sig_func signal(int sig, _sig_func handler);
```

Description

signal() defines how the application responds to the sig signal. The various responses are given by the table below.

Table 19-12. Input Signal Responses

Handler	Response to the Signal
SIG_IGN	Ignores the signal
SIG_DFL	The default response (HALT).
a function	Calls the function with sig as parameter.

The signal handling function is defined as:

```
typedef void (*_sig_func)(int sig);
```

Raise the signal using the raise() function. Before the handler is called, the response is reset to SIG_DFL.

In the CodeWarrior IDE, there are only two signals: SIGABRT indicates an abnormal program termination, and SIGTERM indicates a normal program termination.

Return

If signal succeeds, it returns the previous response for the signal; otherwise it returns SIG_ERR and sets errno to a positive non-zero value.

19.91 sin() and sinf()

Syntax

```
#include <math.h >

double sin(double x);

float sinf(float x);
```

Description

`sin()` computes the sine of `x`.

Return

The sine `sin(x)` of `x` in radians.

See also

asin() and asinf()

acos() and acosf()

atan() and atanf()

atan2() and atan2f()

cos() and cosf()

tan() and tanf()

19.92 sinh() and sinhf()

Syntax

```
#include <math.h >

double sinh(double x);

float sinhf(float x);
```

Description

`sinh()` computes the hyperbolic sine of `x`.

Return

The hyperbolic sine `sinh(x)` of `x`. If it fails because the value is too large, it returns infinity with the same sign as `x` and sets `errno` to `ERANGE`.

See also

asin() and asinf()

19.93 sprintf()

Syntax

```
#include <stdio.h>


int sprintf(char *s, const char *format, ...);
```

Description

`sprintf()` writes formatted output to string `s`. It evaluates the arguments, converts them according to format, and writes the result to `s`, terminated with a zero character.

The format string contains the text to be printed. Any character sequence in format starting with % is a format specifier that is replaced by the corresponding argument. The first format specifier is replaced with the first argument after format, the second format specifier by the second argument, and so on.

A format specifier has the form:

```
FormatSpec      = % {Format}[Width][Precision]

                    [Length]Conversion
```

where:

```
Format          = -|+|<a blank>|#|0
```

The format defines justification and sign information (the latter only for numerical arguments). A - left justifies the output, a + forces output of the sign, and a blank outputs a blank if the number is positive and a - if it is negative. The following table describes the effect of # depending on the conversion character.

Table 19-13. Effect of # in the Format Specification

Conversion	Effect of #
e, E, f	Always prints the value of the argument with decimal point, even if there are no fractional digits.
g, G	As above, but also appends zeroes to the fraction until the specified width is reached.
o	Prints a zero before the number to indicate an octal value.
x, X	Prints 0x (if the conversion is x) or 0X (if it is X) before the number to indicate a hexadecimal value.
Others	Undefined

A 0 as format specifier adds leading zeroes to the number until the desired width is reached, if the conversion character specifies a numerical argument.

If both " " (blank) and + are given, only + is active; if both 0 and - are specified, only - is active. If there is a precision specification for integral conversions, 0 is ignored.

- `Width = *|Number|0Number`

  Number defines the minimum field width into which the output is to be put. If the argument is smaller, the space is filled as defined by the format characters.

  `0Number` is the same as above, but uses zeroes instead of blanks.

  If a * is given, the field width is taken from the next argument, which must be a number. If that number is negative, the output is left-justified.

- `Precision = [Number|*]`

  The effect of the precision specification depends on the conversion character.

Table 19-14. Effect of the Precision Specification

Conversion	Precision
d, i, o, u, x, X	The minimum number of digits to print.
e, E, f	The number of fractional digits to print.
g, G	The maximum number of significant digits to print.
s	The maximum number of characters to print.
Others	Undefined.

If the precision specifier is *, the precision is taken from the next argument, which must be an int. If that value is negative, the precision is ignored.

- `Length = h|l|L`

  A length specifier tells `sprintf()` what type the argument has. The first two length specifiers can be used in connection with all conversion characters for integral numbers. `h` defines `short`; `l` defines `long`. Use specifier `L` in conjunction with the conversion characters for floating point numbers to specify `long double`.

- `Conversion = c|d|e|E|f|g|G|i|n|o|p|s|u|x|X|%`

These conversion characters have the following meanings:

Table 19-15. Meaning of the Conversion Characters

Conversion	Description
`c`	Converts the `int` argument to `unsigned char`; prints the resulting character
`d, i`	Prints an `int` argument
`e, E`	Argument must be a `double`. Prints in the form `[-]d.ddde ±dd` (scientific notation). The precision determines the number of fractional digits, the digit to the left of the decimal is ¦ 0 unless the argument is 0.0. The default precision is 6 digits. If the precision is zero and the format specifier `#` is not given, no decimal point prints. The exponent always has at least two digits; the conversion character prints just before the exponent.
`f`	Argument must be a `double`. Prints in the form `[-]ddd.ddd`. See above. If the decimal point prints, there is at least one digit to the left of it.
`g, G`	Argument must be a `double`. `sprintf()` chooses either format `f` or `e` (or `E` if `G` is given), depending on the magnitude of the value. Uses scientific notation only if the exponent is < -4 or greater than or equal to the precision.
`n`	Argument must be a pointer to an `int`. `sprintf()` writes the number of characters written so far to that address. If `n` is used together with length specifier `h` or `l`, the argument must be a pointer to a `short int` or a `long int`.
`o`	Prints the argument, which must be an `unsigned int`, in octal notation.
`p`	Argument must be a pointer; prints its value in hexadecimal notation.
`s`	Argument must be a `char *`; `sprintf()` writes the string.
`u`	Writes the argument, which must be an `unsigned int`, in decimal notation.
`x, X`	Writes the argument, which must be an `unsigned int`, in hexadecimal notation. `x` uses lower case letters `a` to `f`, while `X` uses upper case letters.
`%`	Prints a `%` sign. Give as `%%` only.

Conversion characters for integral types are `d`, `i`, `o`, `u`, `x`, and `X`; for floating point types `e`, `E`, `f`, `g`, and `G`.

If `sprintf()` finds an incorrect format specification, it stops processing, terminates the string with a zero character, and returns successfully.

NOTE

Floating point support increases the `sprintf` size considerably, and therefore the define `LIBDEF_PRINTF_FLOATING` exists. Set `LIBDEF_PRINTF_FLOATING` if no floating point support is used. Some targets contain special libraries without floating point support.

Return

The number of characters written to `s`.

See also

sscanf()

19.94 sqrt() and sqrtf()

Syntax

```
#include <math.h >


double sqrt(double x);


float  sqrtf(float x);
```

Description

`sqrt()` computes the square root of `x`.

Return

The square root of `x`. If `x` is negative, it returns 0 and sets `errno` to `EDOM`.

See also

pow() and powf()

19.95 srand()

Syntax

```
#include <stdlib.h >


void srand(unsigned int seed)


;
```

Description

`srand()` initializes the seed of the random number generator. The default seed is `1`.

Return

See also

rand()

19.96 sscanf()

Syntax

```
#include <stdio.h>


int sscanf(const char *s, const char *format, ...);
```

Description

`sscanf()` scans string `s` according to the given format, storing the values in the given parameters. The format specifiers in the format tell `sscanf()` what to expect next. A format specifier has the format:

- `FormatSpec = %[Flag][Width][Size]Conversion`

  where:

  - `Flag = *`

    If the % sign, which starts a format specification, is followed by a *, the scanned value is not assigned to the corresponding parameter.

  - `Width = Number`

    Specifies the maximum number of characters to read when scanning the value. Scanning also stops if white space or a character not matching the expected syntax is reached.

  - `Size = h|l|L`

    Specifies the size of the argument to read. The following table describes the meaning of the size argument.

Table 19-16. Relationship of the Size Parameter with Allowable Conversions and Types

Size	Allowable Conversions	Parameter Type
h	d, i, n	short int * (instead of int *)
h	o, u, x, X	unsigned short int * (instead of unsigned int *)
l	d, i, n	long int * (instead of int *)
l	o, u, x, X	unsigned long int * (instead of unsigned int *)
l	e, E, f, g, G	double * (instead of float *)
L	e, E, f, g, G	long double * (instead of float *)

  - `Conversion = c|d|e|E|f|g|G|i|n|o|p|s|u|x|X|%|Range`

These conversion characters tell `sscanf()` what to read and how to store it in a parameter. Their meaning is shown in the following table.

Table 19-17. Actions Taken for Conversions

Conversion	Description
`c`	Reads a string of exactly `width` characters and stores it in the parameter. If no width is given, reads one character. The argument must be a `char *`. The string read is *not* zero-terminated.
`d`	Reads as a decimal number (syntax below) and stores in the parameter. The parameter must be a pointer to an integral type.
`i`	Reads as `d` (above), but also reads octal and hexadecimal numbers (syntax below).
`e`, `E`, `f`, `g`, or `G`	Reads a floating-point number (syntax below). The parameter must be a pointer to a floating-point type.
`n`	The argument must be a pointer to an `int`. `sscanf()` writes the number of characters read so far to that address. If `n` is used together with length specifier `h` or `l`, the argument must be a pointer to a `shortint` or a `longint`.
`o`	Reads an octal number (syntax below). The parameter must be a pointer to an integral type.
`p`	Reads a pointer in the same format as sprintf() prints it. The parameter must be a `void **`.
`s`	Reads a character string up to the next white space character or at most `width` characters. The string is zero-terminated. The argument must be of type `char *`.
`u`	Reads as `d` (above), but the parameter must be a pointer to an unsigned integral type.
`x`, `X`	Reads as `u` (above), but reads a hexadecimal number.
`%`	Skips a `%` sign in the input. Give only as `%%`.

```
Range = [[^]List]

List = Element {Element}

Element = <any char> [-<any char>]
```

You can also use a `scan` set to read a character string that either contains only the given characters or contains only characters not in the set. A scan set always is bracketed by left and right brackets. If the first character in the set is ^, the set is inverted (that is, only characters *not* in the set are allowed). You can specify whole character ranges, (for example, `A-Z` specifies all upper-case letters). To include a right bracket in the scan set, it must be the first element in the list; a dash (`-`) must be either the first or the last element. To include a `^` in the list (instead of indicating an inverted list) ensure that the `^` is not the first character after the left bracket.

Some examples are:

`[A-Za-z]` Allows all upper- and lower-case characters.

`[^A-Z]` Allows any character that is not an upper-case character.

`[]abc]` Allows], a, b and c.

`[^]abc]` Allows any character except], a, b and c.

`[-abc]` Allows -, a, b and c.

A white space in the format string skips all white space characters up to the next non-white-space character. Any other character in the format must be exactly matched by the input; otherwise `sscanf()` stops scanning.

The syntax for numbers as scanned by `sscanf()` is the following:

```
Number = FloatNumber|IntNumber

IntNumber = DecNumber|OctNumber|HexNumber

DecNumber = SignDigit{Digit}

OctNumber = Sign0{OctDigit}

HexNumber = 0(x|X) HexDigit {HexDigit}

FloatNumber = Sign {Digit} [.{Digit}] [Exponent]

Exponent = (e|E)DecNumber
```

```
OctDigit = 0|1|2|3|4|5|6|7

Digit = OctDigit|8|9

HexDigit = Digit |A|B|C|D|E|F|a|b|c|d|e|f
```

Return

EOF, if s is NULL; otherwise it returns the number of arguments filled in.

NOTE

If sscanf() finds an illegal input (that is, not matching the required syntax), it simply stops scanning and returns successfully.

19.97 strcat()

Syntax

```
#include <string.h >

char *strcat(char *p, const char *q);
```

Description

strcat() appends string q to the end of string p. Both strings and the resulting concatenation are zero-terminated.

Return

p

See also

memcpy() and memmove()

strcpy()

strncat()

strncpy()

19.98 strchr()

Syntax

```
#include <string.h >


char *strchr(const char *p, int ch);
```

Description

`strchr()` looks for character `ch` in string `p`. If `ch` is `\0`, the function looks for the end of the string.

Return

A pointer to the character, if found; if there is no such character in `*p`, `NULL` is returned.

See also

memchr()

strrchr()

strstr()

19.99 strcmp()

Syntax

```
#include <string.h >


int strcmp(const char *p, const char *q);
```

Description

`strcmp()` compares the two strings, using the character ordering given by the ASCII character set.

Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

NOTE

The return value of `strcmp()` can be used as a comparison function in bsearch() and qsort().

See also

memcmp()

strcoll()

strncmp()

19.100 strcoll()

Syntax

```
#include <string.h >


int strcoll(const char *p, const char *q);
```

Description

`strcoll()` compares the two strings interpreting them according to the current locale, using the character ordering given by the ASCII character set.

Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

See also

memcmp()

strcpy()

strncmp()

19.101 strcpy()

Syntax

```
#include <string.h >


char *strcpy(char *p, const char *q);
```

Description

`strcpy()` copies string `q` into string p (including the terminating `\0`).

Return

`p`

See also

memcpy() and memmove()

strncpy()

19.102 strcspn()

Syntax

```
#include <string.h >


size_t strcspn(const char *p, const char *q);
```

Description

`strcspn()` searches `p` for the first character that also appears in `q`.

Return

The length of the initial segment of `p` that contains only characters *not* in `q`.

See also

strchr()

strpbrk()

strrchr()

strspn()

19.103 strerror()

Syntax

```
#include <string.h >


char *strerror(int errno);
```

Description

`strerror()` returns an error message appropriate for error number `errno`.

Return

A pointer to the message string.

See also

perror()

19.104 strftime()

Syntax

```
#include <time.h >


size_t strftime(char *s,
```

```
                size_t max,

                const char *format,

                const struct tm *time);
```

Description

`strftime()` converts `time` to a character string `s`. If the conversion results in a string longer than `max` characters (including the terminating `\0`), `strftime()` leaves `s` unchanged and the function returns unsuccessfully. The format string determines the conversion process. This string contains text, which is copied one-to-one to `s`, and format specifiers. Format specifiers always start with `%` sign and are replaced by the following:

Table 19-18. strftime() Output String Content and Format

Format	Replaced With
`%a`	Abbreviated name of the weekday of the current locale (for example, `Fri`)
`%A`	Full name of the weekday of the current locale (for example, `Friday`)
`%b`	Abbreviated name of the month of the current locale (for example, `Feb`)
`%B`	Full name of the month of the current locale (for example, `February`)
`%c`	Date and time in the form given by the current locale.
`%d`	Day of the month in the range from 0 to 31.
`%H`	Hour, in 24-hour-clock format.
`%I`	Hour, in 12-hour-clock format.
`%j`	Day of the year, in the range from 0 to 366.
`%m`	Month, as a decimal number from 0 to 12.
`%M`	Minutes
`%p`	AM/PM specification of a 12-hour clock or equivalent of current locale.
`%S`	Seconds
`%U`	Week number in the range from 0 to 53, with Sunday as the first day of the first week.
`%w`	Day of the week (Sunday = 0, Saturday = 6)
`%W`	Week number in the range from 0 to 53, with Monday as the first day of the first week
`%x`	The date in format given by current locale
`%X`	The time in format given by current locale

Table continues on the next page...

Table 19-18. strftime() Output String Content and Format (continued)

Format	Replaced With
%y	The year in short format (for example, 93)
%Y	The year, including the century (for example, 1993)
%Z	The time zone, if it can be determined.
%%	A single % sign

Return

Returns zero if the resulting string has more than max characters; otherwise returns the length of the created string.

See also

mktime()

setlocale()

time()

19.105 strlen()

Syntax

```
#include <string.h >


size_t strlen(const char *s);
```

Description

strlen() returns the number of characters in string s.

Return

The length of the string.

See also

19.106 strncat()

Syntax

```
#include <string.h >


char *strncat(char *p, const char *q, size_t n);
```

Description

`strncat()` appends string `q` to string `p`. If `q` contains more than `n` characters, only the first `n` characters of `q` are appended to `p`. The two strings and the result all are zero-terminated.

Return

`p`

See also

strcat()

19.107 strncmp()

Syntax

```
#include <string.h >


char *strncmp(char *p, const char *q, size_t n);
```

Description

`strncmp()` compares at most the first `n` characters of the two strings.

Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal; or a positive integer if `p` is greater than `q`.

See also

memcmp()

strcmp()

19.108 strncpy()

Syntax

```
#include <string.h >

char *strncpy(char *p, const char *q, size_t n);
```

Description

`strncpy()` copies at most the first `n` characters of string `q` to string `p`, overwriting `p`'s previous contents. If `q` contains less than `n` characters, `strncpy()` appends a `\0`.

Return

`p`

See also

memcpy()

strcpy()

19.109 strpbrk()

Syntax

```
#include <string.h >

char *strpbrk(const char *p, const char *q);
```

Description

`strpbrk()` searches for the first character in `p` that also appears in `q`.

Return

`NULL`, if there is no such character in `p`; a pointer to the character otherwise.

See also

strchr()

strcspn()

strrchr()

strspn()

19.110 strrchr()

Syntax

```
#include <string.h >


char *strrchr(const char *s, int c);
```

Description

`strpbrk()` searches for the last occurrence of character `ch` in `s`.

Return

`NULL`, if there is no such character in `p`; a pointer to the character otherwise.

See also

strchr()

strcspn()

strpbrk()

strspn()

19.111 strspn()

Syntax

```
#include <string.h >


size_t strspn(const char *p, const char *q);
```

Description

`strspn()` returns the length of the initial part of `p` that contains only characters also appearing in `q`.

Return

The position of the first character in `p` that is not in `q`.

See also

strchr()

strcspn()

strpbrk()

strrchr()

19.112 strstr()

Syntax

```
#include <string.h >


char *strstr(const char *p, const char *q);
```

Description

`strstr()` looks for substring `q` appearing in string `p`.

Return

A pointer to the beginning of the first occurrence of string `q` in `p`, or `NULL`, if `q` does not appear in `p`.

See also

strchr()

strcspn()

strpbrk()

strrchr()

strspn()

19.113 strtod()

Syntax

```
#include <stdlib.h >


double strtod(const char *s, char **end);
```

Description

`strtod()` converts string `s` into a floating point number, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax and returns a pointer to that character in `*end`. `strtod()` accepts the following number format:

```
FloatNum = Sign{Digit}[.{Digit}][Exp]

Sign = [+|-]

Exp = (e|E)SignDigit{Digit}

Digit = <any decimal digit from 0 to 9>
```

Return

The floating point number read. If an underflow occurred, `strtod()` returns `0.0`. If the value causes an overflow, `strtod()` returns `HUGE_VAL`. In both cases, `strtod()` sets `errno` to `ERANGE`.

See also

atof()

scanf()

strtol()

strtoul()

19.114 strtok()

Syntax

```
#include <string.h >


char *strtok(char *p, const char *q);
```

Description

`strtok()` breaks the string `p` into tokens, separated by at least one character appearing in `q`. The first time, call `strtok()` using the original string as the first parameter. Afterwards, pass `NULL` as first parameter: `strtok()` continues from its previous stopping position. `strtok()` saves the string `p` unless it is `NULL`.

NOTE

This function is not re-entrant because it uses a global variable for saving string `p`. ANSI defines this function in this way.

Return

A pointer to the token found, or `NULL`, if no token was found.

See also

strchr()

strcspn()

strpbrk()

strrchr()

strspn()

strstr()

19.115 strtol()

Syntax

```
#include <stdlib.h >

long strtol(const char *s, char **end, int base);
```

Description

`strtol()` converts string `s` into a `longint` of base `base`, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in `*end`.

`strtol()` accepts the following number format:

```
Int_Number =

Dec_Number|Oct_Number|Hex_Number|Other_Number

Dec_Number = SignDigit{Digit}

Oct_Number = Sign0{OctDigit}

Hex_Number = 0(x|X)Hex_Digit{Hex_Digit}

Other_Number = SignOther_Digit{Other_Digit}

Oct_Digit = 0|1|2|3|4|5|6|7

Digit = Oct_Digit|8|9

Hex_Digit = Digit |A|B|C|D|E|F|a|b|c|d|e|f

Other_Digit = Hex_Digit|<any char between G and Z>|<any char
between g and z>
```

The base must be 0 or in the range from 2 to 36. If it is between 2 and 36, `strtol()` converts a number in that base (digits larger than 9 are represented by upper or lower case characters from `A` to `Z`). If base is zero, the function uses the prefix to find the base. If the prefix is `0`, `strtol()` assumes base 8 (octal). If it is `0x` or `0X`, `strtol()` assumes base 16 (hexadecimal). Any other prefixes make `strtol()` scan a decimal number.

Return

The number read. If no number is found, returns zero; if the value is smaller than `LONG_MIN` or larger than `LONG_MAX`, returns `LONG_MIN` or `LONG_MAX` and sets `errno` to `ERANGE`.

See also

atoi()

atol()

scanf()

strtod()

strtoul()

19.116 strtoul()

Syntax

```
#include <stdlib.h >


unsigned long strtoul(const char *s,

                      char **end,

                      int base);
```

Description

strtoul() converts string s into an unsigned long int of base base, ignoring any white space at the beginning of s. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in *end. strtoul() accepts the same number format as strtol() except that the negative sign is not allowed, and neither are the possible values for base.

Return

The number read. If no number is found, returns zero; if the value is larger than ULONG_MAX, returns ULONG_MAX and sets errno to ERANGE.

See also

atoi()

atol()

scanf()

strtod()

strtol()

19.117 strxfrm()

Syntax

```
#include <string.h >


size_t strxfrm(char *p, const char *q, size_t n);
```

Description

strxfrm() transforms string q according to the current locale, such that the comparison of two strings converted with strxfrm() using strcmp() yields the same result as a comparison using strcoll(). If the resulting string is longer than n characters, strxfrm() leaves p unchanged.

Return

The length of the converted string.

See also

setlocale()

strcmp()

strcoll()

19.118 system()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <string.h >


int system(const char *cmd);
```

Description

`system()` executes the command line `cmd`.

Return

Zero

See also

19.119 tan() and tanf()

Syntax

```
#include <math.h >


double tan(double x);


float  tanf(float x);
```

Description

`tan()` computes the tangent of `x`. Give `x` in radians.

Return

`tan(x)`. If `x` is an odd multiple of `Pi/2`, it returns infinity and sets `errno` to `EDOM`.

See also

acos() and acosf()

asin() and asinf()

atan() and atanf()

atan2() and atan2f()

cosh() and coshf()

sin() and sinf()

tan() and tanf()

19.120 tanh() and tanhf()

Syntax

```
#include <math.h >


double tanh(double x);


float  tanhf(float x);
```

Description

`tanh()` computes the hyperbolic tangent of `x`.

Return

```
tanh(x)
```

See also

atan() and atanf()

atan2() and atan2f()

cosh() and coshf()

sin() and sinf()

tan() and tanf()

19.121 time()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <time.h >


time_t time(time_t *timer);
```

Description

`time()` gets the current calendar time. If `timer` is not `NULL`, the current calendar time is assigned to `timer`.

Return

The current calendar time.

See also

clock()

mktime()

strftime()

19.122 tmpfile()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


FILE *tmpfile(void);
```

Description

`tmpfile()` creates a new temporary file using mode `wb+`. Temporary files are deleted automatically when closed or the application ends.

Return

A pointer to the file descriptor if the file is created; `NULL` otherwise.

See also

fopen()

tmpnam()

19.123 tmpnam()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


char *tmpnam(char *s);
```

Description

`tmpnam()` creates a new unique filename. If `s` is not `NULL`, `tmpnam()` assigns the new name to `s`.

Return

A unique filename

See also

tmpfile()

19.124 tolower()

Syntax

```
#include <ctype.h >


int tolower(int ch);
```

Description

`tolower()` converts any upper-case character in the range from `A` to `Z` into a lower-case character from `a` to `z`.

Return

If `ch` is an upper-case character, returns the corresponding lower-case letter. Otherwise, returns `ch` (unchanged).

See also

islower()

isupper()

toupper()

19.125 toupper()

Syntax

```
#include <ctype.h >


int toupper(int ch);
```

Description

`toupper()` converts any lower-case character in the range from `a` to `z` into an upper-case character from `A` to `Z`.

Return

If `ch` is a lower-case character, returns the corresponding upper-case letter. Otherwise, returns `ch` (unchanged).

See also

islower()

isupper()

tolower()

19.126 ungetc()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >


int ungetc(int ch, FILE *f)


;
```

Description

`ungetc()` pushes the single character `ch` back onto the input stream `f`. The next read from `f` reads that character.

Return

`ch`

See also

fgets()

fopen()

getc()

getchar()

19.127 va_arg(), va_end(), and va_start()

Syntax

```
#include <stdarg.h >


void va_start(va_list args,
param);


type va_arg(va_list args,
type);


void va_end(va_list args);
```

Description

These macros can be used to get the parameters in an open parameter list. Calls to `va_arg()` get a parameter of the given type. The following example demonstrates the use of `va_arg()`:

```
void my_func(char *s, ...) {

 va_list args;

 int i;

 char *q;
```

```
  va_start(args, s);

  /* First call to 'va_arg' gets the first arg. */

  i = va_arg (args, int);

  /* Second call gets the second argument. */

  q = va_arg(args, char *);

  ...

  va_end (args);

}
```

19.128 vfprintf(), vprintf(), and vsprintf()

This is a file I/O function, also hardware dependent. It is not implemented in this Compiler.

Syntax

```
#include <stdio.h >

int vfprintf(FILE *f,

             const char *format,

             va_list args);
```

```
int vprintf(const char *format, va_list args);


int vsprintf(char *s,


             const char *format,


             va_list args);
```

Description

These functions are the same as fprintf(), printf(), and sprintf(), except that they take a `va_list` instead of an open parameter list as argument.

For a detailed format description see `sprintf()`.

NOTE

Only `vsprintf()` is implemented, because the other two functions depend on the actual setup and environment of the target.

Return

The number of characters written, if successful; a negative number otherwise.

See also

fprintf()

printf()

sprintf()

va_arg(), va_end(), and va_start()

19.129 wctomb()

Syntax

```
#include <stdlib.h >
```

```
int wctomb(char *s, wchar_t wchar);
```

Description

`wctomb()` converts `wchar` to a multi-byte character, stores that character in `s`, and returns the length of `s` in bytes.

Return

The length of `s` in bytes after the conversion.

See also

wcstombs()

19.130 wcstombs()

This function is Hardware-specific implementation. It is not implemented in this Compiler.

Syntax

```
#include <stdlib.h >


int wcstombs(char *s, const wchar_t *ws, size_t n);
```

Description

`wcstombs()` converts the first `n` wide character codes in `ws` to multi-byte characters, stores them in `s`, and returns the number of wide characters converted.

Return

The number of wide characters converted.

See also

wctomb()

Chapter 20
Appendices

This manual includes the following appendices:

- Porting Tips and FAQs : Hints about EBNF notation used by the linker and about porting applications from other Compiler vendors to this Compiler
- Global Configuration File Entries : Describes the entries in the `mcutools.ini` file
- Local Configuration File Entries : Describes the entries in the `project.ini` file
- Known C++ Issues in the HC(S)08 Compilers : Describes unsupported features and known issues with the C++ Compiler
- Banked Memory Support : Describes unsupported features and known issues with the C++ Compiler
- Compiler Messages : Documentation on compiler messages.

Chapter 21
Porting Tips and FAQs

This appendix describes some FAQs and provides tips on EBNF syntax and porting the application from a different tool vendor.

This chapter covers the following topics:

21.1 Migration Hints

This section describes the differences between this compiler and the compilers of other vendors. It also provides information about porting sources and methods of adapting them.

The topics covered are as follows:

- @tiny and @far Qualifier for Variables
- Arrays with Unknown Size
- Missing Prototype
- _asm("sequence")
- Recursive Comments
- Interrupt Function, @interrupt
- Defining Interrupt Functions

21.1.1 Porting from Cosmic

If your current application is written for Cosmic compilers, there are some special things to consider. The topics covered in this section are as follows:

- Getting Started
- Cosmic Compatibility Mode Switch
- Assembly Equates
- Inline Assembly Identifiers
- Pragma Sections
- Inline Assembly Constants
- Inline Assembly and Index Calculation
- Inline Assembly and Tabs
- Inline Assembly and Operators
- @interrupt
- Inline Assembly and Conditional Blocks
- Compiler Warnings
- Linker *.prm File (for the Cosmic compiler) and Linker *.prm File (for the HC(S)08 Compiler)

21.1.1.1 Getting Started

The best way is to create a new project, either using the New Bareboard Project wizard (**File > New Bareboard Project**) or with a stationery template. This sets up a project for you, including all the default options and library files. Then add the existing files used for Cosmic to the project (for example, through drag and drop from the Windows Explorer, or right-clicking in the CodeWarrior Projects view and selecting **Add Files**). Ensure that the new project uses the same memory model and CPU type as the Cosmic project.

21.1.1.2 Cosmic Compatibility Mode Switch

The latest compiler offers a Cosmic compatibility mode switch (-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers). When you enable this compiler option, the compiler accepts most Cosmic constructs.

21.1.1.3 Assembly Equates

For the Cosmic compiler, you use `equ` to define equates for the inline assembly. To use an equate or value in C as well, you use `#define` to define it. For this compiler, only `#define` is needed for C and for inline assembly (refer the following listing). The `equ` directive is not supported in normal C code.

Listing: EQU Directive Example

```
#ifdef  __MWERKS__
#define CLKSRC_B   0x00 /*; Clock source */
#else
  CLKSRC_B : equ   $00    ; Clock source
#endif
```

21.1.1.4 Inline Assembly Identifiers

For the Cosmic compiler, you need to place an underscore (_) in front of each identifier, but for this compiler you can use the same name both for C and inline assembly. In addition, to use the address of a variable, place a `@` in front of variables for better type-safety with this compiler. Use a conditional block like the one below in the following listing, to compensate for the differences in compilers.

Listing: Using a Conditional Block

```
#ifdef __MWERKS__
  ldx @myVariable,x
  jsr MyFunction
```

```
#else
  ldx _myVariable,x
  jsr _MyFunction
#endif
```

You can also use macros to cope with the compiler differences. The following listing shows the preferred method.

Listing: Using a Macro

```
#ifdef __MWERKS__
  #define USCR(ident)  ident
  #define USCRA(ident) @ ident
#else /* for COSMIC, add a _ (underscore) to each ident */
  #define USCR(ident)  _##ident
  #define USCRA(ident) _##ident
#endif
```

The source can use the macros:

```
  ldx USCRA(myVariable),x


  jsr USCR(MyFunction)
```

21.1.1.5 Pragma Sections

Cosmic uses the `#pragma section` syntax, while this compiler employs either `#pragma DATA_SEG` or `#pragma CONST_SEG`.

Listing: #pragma DATA_SEG

```
#ifdef __MWERKS__ #pragma DATA_SEG APPLDATA_SEG
#else
#pragma section {APPLDATA}
#endif
```

Listing: #pragma CONST_SEG

```
#ifdef __MWERKS__ #pragma CONST_SEG CONSTVECT_SEG
#else
#pragma section const {CONSTVECT}
#endif
```

Use the segments (in the examples above `CONSTVECT_SEG` and `APPLDATA_SEG`) in the linker `*.prm` file in the `PLACEMENT` block.

21.1.1.6 Inline Assembly Constants

Cosmic uses an assembly constant syntax, whereas this compiler employs the normal C constant syntax (refer the following listing).

Listing: Normal C Constant Syntax

```
#ifdef __MWERKS__
  and 0xF8

#else

  and #$F8

#endif
```

21.1.1.7 Inline Assembly and Index Calculation

The Cosmic compiler uses the + operator to calculate offsets into arrays. For the CodeWarrior compiler, use a colon (:) instead:

Listing: Using a Colon for Offset

```
ldx array:7
#else

  ldx array+7

#endif
```

21.1.1.8 Inline Assembly and Tabs

The Cosmic compiler lets you use TAB characters in normal C strings (surrounded by double quotes):

```
  asm("This string contains hidden tabs!");
```

Because the compiler rejects hidden tab characters in C strings according to the ANSI-C standard, you must remove the tab characters from such strings.

21.1.1.9 Inline Assembly and Operators

The Cosmic compiler and this compiler's inline assembly may not support the same amount or level of operators. But in most cases it is simple to rewrite or transform them (refer the following listing).

Listing: Compensating for Different Operators between Different Compilers

```
#ifdef __MWERKS__
  ldx #(BOFFIE + WUPIE) ; enable Interrupts
#else
  ldx #(BOFFIE | WUPIE) ; enable Interrupts
#endif
#ifdef __MWERKS__
  lda    #(_TxBuf2+Data0)
  ldx   #((_TxBuf2+Data0) / 256)
#else
  lda   #((_TxBuf2+Data0) & $ff)
  ldx   #(((_TxBuf2+Data0) >> 8) & $ff)
#endif
```

21.1.1.10 @interrupt

The Cosmic compiler uses the `@interrupt` syntax, whereas this compiler employs the `interrupt` syntax. To keep the source base portable, you can use a macro (for example, in a main header file, which selects the correct syntax depending on the compiler used).

Listing: interrupt Syntax

```
/* place the following in a header file: */
#ifdef __MWERKS__

  #define INTERRUPT interrupt

#else

  #define INTERRUPT @interrupt

#endif

/* now for each @interrupt we use the INTERRUPT macro: */

void INTERRUPT myISRFunction(void) { ....
```

21.1.1.11 Inline Assembly and Conditional Blocks

In most cases, the (-Ccx: Cosmic Compatibility Mode for Space Modifiers and Interrupt Handlers) handles the `#asm` blocks used in the Cosmic compiler's inline assembly code Cosmic compatibility switch. However, if you use #asm with conditional blocks like `#ifdef` or `#if`, then the C parser may not accept it (refer the following listing).

Listing: Using Conditional Blocks without asm Block Markers ({ and })

```
void myfun(void) {
  #asm

    nop

#if 1

  #endasm

  myfun();

  #asm

    #endif

    nop

  #endasm

}
```

In such cases, the `#asm` and `#endasm` must be ported to `asm { ` and `}` block markers (refer the following listing).

Listing: Using Conditional Blocks with asm Block Markers ({ and })

```
void myfun(void) {
  asm { // asm #1
```

```
    nop
#if 1
  } // end of asm #1
  myfun();
  asm { // asm #2
#endif
    nop
  } // end of asm #2
}
```

21.1.1.12 Compiler Warnings

Check the warnings produced by the compiler carefully. The Cosmic compiler does not warn about many cases where your application code may contain a bug. Later on you can switch the warning messages off if they issues they highlight are unimportant (for example, use the `-W2: Do not Print INFORMATION or WARNING Messages` option or use #pragma MESSAGE: Message Setting in the source code).

21.1.1.13 Linker *.prm File (for the Cosmic compiler) and Linker *.prm File (for the HC(S)08 Compiler)

The Cosmic compiler uses a `*.prm` file with a special syntax for the linker. This compiler uses a linker parameter file with a `*.prm` file extension. The syntax is not the same format, but most things are straightforward to port. For this compiler, you must declare the RAM or ROM areas in the `SEGMENTS ... END` block and place the sections into the `SEGMENTS` in the `PLACEMENT...END` block.

Make sure that you use all segments declared in your application (through `#pragma DATA_SEG`, `#pragma CONST_SEG`, and `#pragma CODE_SEG`) in the `PLACEMENT` block of the Linker prm file.

Check the linker warnings or errors carefully. They may indicate needed adjustments or corrections in your application. For example, you may have allocated the vectors in the linker `*.prm` file (using `VECTOR` or `ADDRESS` syntax) and also allocated them in the application itself (with the `#pragma CONST_SEG` or with the `@address` syntax). Allocating objects twice is an error, so allocate these objects one way or the other, but not both.

Consult the map file produced by the linker to ensure that everything is allocated correctly.

Remember that the linker is a smart linker. This means the linker does not link unused or unreferenced objects to the application (the Cosmic linker may link unused or unreferenced objects). The application may need to link to these objects for some reason even though the linker does not. To link objects to the application even when unused, use the `ENTRIES ... END` block in the Linker `*.prm` file:

```
  ENTRIES /* the following objects or variables need to be linked even if not referenced by
the application */


  _vectab ApplHeader FlashEraseTable


  END
```

21.1.2 Allocation of Bitfields

Bitfield allocation depends on the compiler. Some compilers allocate the bits from right (least significant byte) to left (most significant byte), and others allocate from left to right. Also, some compilers implement alignment and byte or word crossing of bitfields inconsistently. Solutions include:

- Check the different allocation strategies,
- Find out if an option exists to change the compiler allocation strategy, or
- Use the compiler defines to maintain source portability:
  - `__BITFIELD_LSBIT_FIRST__`
  - `__BITFIELD_MSBIT_FIRST__`
  - `__BITFIELD_LSBYTE_FIRST__`
  - `__BITFIELD_MSBYTE_FIRST__`
  - `__BITFIELD_LSWORD_FIRST__`
  - `__BITFIELD_MSWORD_FIRST__`
  - `__BITFIELD_TYPE_SIZE_REDUCTION__`
  - `__BITFIELD_NO_TYPE_SIZE_REDUCTION__`

21.1.3 Type Sizes and Sign of char

Carefully check the type sizes used by your compiler uses. Different compilers implement the sizes for the standard types (`char`, `short`, `int`, `long`, `float`, or `double`) differently. For instance, the size for `int` is 16 bits for some compilers and 32 bits for others.

If necessary, change the default type settings with the -T: Flexible Type Management option.

Also, the sign of `plain char` is inconsistent across compilers. If the software program requires that `char` be signed or unsigned, either change all `plainchar` types to the signed or unsigned types or change the sign of char with the -T option.

21.1.4 @bool Qualifier

Some compiler vendors provide the special keyword `@bool` to specify that a function returns a boolean value:

```
@bool int myfun(void);
```

Because this compiler does not support this special keyword, remove `@bool` or use a define such as this:

```
#define _BOOL /*@bool*/


_BOOL int myfun(void);
```

21.1.5 @tiny and @far Qualifier for Variables

Some compiler vendors provide special keywords to place variables in absolute locations. In ANSI-C, you can express such absolute locations as constant pointers:

```
#ifdef __HIWARE__
```

```
  #define REG_PTB (*(volatile char*)(0x01))


#else /* other compiler vendors use non-ANSI features */


  @tiny volatile  char REG_PTB    @0x01; /* port B */


#endif
```

The Compiler does not need the @tiny qualifier directly. The Compiler is smart enough to take the right addressing mode depending on the address:

```
/* compiler uses the correct addressing mode */


volatile char REG_PTB @0x01;
```

21.1.6 Arrays with Unknown Size

Some compilers accept the following non-ANSI compliant statement to declare an array with an unknown size:

```
extern char buf[0];
```

However, the compiler issues an error message for this because an object with size zero (even if declared as extern) is illegal. Use the legal version:

```
extern char buf[];
```

21.1.7 Missing Prototype

Many compilers accept a function-call usage without a prototype. This compiler issues a warning for this. However if the prototype of a function with open arguments is missing or this function is called with a different number of arguments, this is clearly an error:

```
printf(
"hello world!"); // compiler assumes void



printf(char*);



// error, argument number mismatch!



printf(
"hello %s!", "world");
```

To avoid such programming bugs use the -Wpd: Error for Implicit Parameter Declaration compiler option and always include or provide a prototype.

21.1.8 _asm("sequence")

Some compilers use `_asm(`< *string*> `)` to write inline assembly code in normal C source code: `_asm(`< `nop`> `)`;

Rewrite this with `asm` or `asm {}: asm nop;`

21.1.9 Recursive Comments

Some compilers accept recursive comments without any warnings. This Compiler issues a warning for each such recursive comment:

```
/* this is a recursive comment /*



  int a;
```

```
  /* */
```

The Compiler treats the above source as one single comment, so the definition of `a` is inside the comment. That is, the Compiler treats everything between the first opening comment (`/*`) until the closing comment token (`*/`) as a comment. Change any existing recursive comments.

21.1.10 Interrupt Function, @interrupt

Mark interrupt functions with #pragma TRAP_PROC or use the interrupt keyword (refer the following listing).

Listing: Using the TRAP_PROC Pragma with an Interrupt Function

```
#ifdef __HIWARE__
  #pragma
TRAP_PROC

  void MyTrapProc(void)

#else /* other compiler-vendor non-ANSI declaration of interrupt

         function */

  @interrupt void MyTrapProc(void)

#endif

{

  /* code follows here */

}
```

21.1.11 Defining Interrupt Functions

This section discusses some important topics related to interrupt function handling:

- Defining an Interrupt Function
- Initializing the Vector Table
- Placing an Interrupt Function in a Special Section

21.1.11.1 Defining an Interrupt Function

The compiler provides two ways to define an interrupt function:

- Using the TRAP_PROC Pragma
- Using the Interrupt Keyword

21.1.11.1.1 Using the TRAP_PROC Pragma

The TRAP_PROC pragma informs the compiler that the following function is an interrupt function (refer the following listing). In that case, the compiler should terminate the function by a special interrupt return sequence (for many processors, an RTI instead of an RTS).

Listing: Example of Using the TRAP_PROC Pragma

```
#pragma TRAP_PROC
void INCcount(void) {

  tcount++;

}
```

21.1.11.1.2 Using the Interrupt Keyword

The interrupt keyword is non-standard ANSI-C and therefore is not supported by all ANSI-C compiler vendors. In the same way, the syntax for the usage of this keyword may change between different compilers. The keyword interrupt informs the compiler that the following function is an interrupt function (refer the following listing).

Listing: Example of Using the Interrupt Keyword

```
interrupt void INCcount(void) {
  tcount++;

}
```

21.1.11.2 Initializing the Vector Table

Once you write the interrupt function code, you must associate this function with an interrupt vector. Do this by initializing the vector table. You can initialize the vector table in the following ways:

- Using the Linker Commands `VECTOR ADDRESS` or `VECTOR` in the PRM file
- Using the Interrupt Keyword

21.1.11.2.1 Using the Linker Commands

The Linker provides two commands to initialize the vector table: `VECTOR ADDRESS` or `VECTOR`. Use the `VECTOR ADDRESS` command to write the address of a function at a specific address in the vector table.

To enter the address of the `INCcount()` function at address 0x8A, insert the following command in the application's PRM file.

Listing: Using the VECTOR ADDRESS Command

```
VECTOR ADDRESS 0x8A INCcount
```

Use the VECTOR command to associate a function with a specific vector, identified by its number. The mapping from the vector number is specific to the target.

To associate the address of the `INCcount()` function with the vector number 42, insert the following command in the application's PRM file.

Listing: Using the VECTOR Command

```
VECTOR 42 INCcount
```

21.1.11.2.2 Using the Interrupt Keyword

When using the interrupt keyword, you may directly associate your interrupt function with a vector number in the ANSI C-source code file. For that purpose, specify the vector number next to the keyword interrupt.

To associate the address of the `INCcount()` function with the vector number 42, define the function as in the following listing.

Listing: Definition of the INCcount() Interrupt Function

```
interrupt 42 void INCcount(void) {
int card1;

tcount++;

}
```

21.1.11.3 Placing an Interrupt Function in a Special Section

For all targets supporting paging, allocate the interrupt function in an area that is accessible all the time. You can do this by placing the interrupt function in a specific segment, as listed below:

- Defining a Function in a Specific Segment
- Allocating a Segment in Specific Memory

21.1.11.3.1 Defining a Function in a Specific Segment

To define a function in a specific segment, use the CODE_SEG pragma, as shown in the following listing.

Listing: Defining a Function in a Specific Segment

```
/* This function is defined in segment `int_Function'*/
#pragma CODE_SEG Int_Function

#pragma TRAP_PROC

void INCcount(void) {

  tcount++;

}

#pragma CODE_SEG DEFAULT /* Back to default code segment.*/
```

21.1.11.3.2 Allocating a Segment in Specific Memory

In the PRM file, you can specify where to allocate each segment defined in your source code. To place a segment in a specific memory area, add the segment name in the PLACEMENT block of your PRM file. Remember the linker is case-sensitive; ensure that your segment name is exact, as shown in the following listing.

Listing: Allocating a Segment in Specific Memory

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

SECTIONS

  INTERRUPT_ROM = READ_ONLY   0x4000 TO  0x5FFF;

  MY_RAM        = READ_WRITE  ....

PLACEMENT

  Int_Function               INTO INTERRUPT_ROM;

  DEFAULT_RAM                INTO MY_RAM;

  ....

END
```

21.2 Protecting Parameters in the OVERLAP Area

Instead of using memory on the stack, some targets may direct compilers to place parameters and local variables into a global memory area called `OVERLAP`. For targets without stack access, there is no other way to handle parameters and local variables.

The example below demonstrates the use of pragma `NO_OVERLAP` and `NO_ENTRY` to protect the `Tim_PresetTimer()` function. Various code sequences, particularly interrupt service routines (ISRs), call `Tim_PresetTimer()`. The parameters passed to `Tim_PresetTimer()` are placed in the overlap area. The following example protects these parameters from overwriting.

```
#include <hidef.h>


extern char timer[];


#pragma NO_OVERLAP


#pragma NO_ENTRY


void Tim_PresetTimer(unsigned char tIndex, unsigned char
PresetValue)
```

```
{

  DisableInterrupts;

  asm {

    STX   tIndex

    STA   PresetValue

  }

  timer[tIndex] = PresetValue;

  EnableInterrupts;

}
```

As an example the following code is given, but it looks similar for other targets:

```
1:  #include <hidef.h>

2:

3:  extern char timer[];

4:
```

```
    5:  #pragma NO_OVERLAP


    6:  #pragma NO_ENTRY


    7:  void Tim_PresetTimer(unsigned char tIndex, unsigned
char


PresetValue)


    8:  {


    9:    DisableInterrupts;


Function: Tim_PresetTimer


Options : -Ix:\chc05\lib\hc05c\include -Lasm=%n.lst


  0000 9b       SEI


   10:    asm {


   11:      STX   tIndex


  0001 cf0000   STX   _Tim_PresetTimerp1


   12:      STA   PresetValue
```

```
0004 c70000    STA   _Tim_PresetTimerp0

 13:   }

 14:    timer[tIndex] = PresetValue;

0007 ce0000    LDX   _Tim_PresetTimerp1

000a c60000    LDA   _Tim_PresetTimerp0

000d d70000    STA   timer,X

 15:    EnableInterrupts;

0010 9a        CLI

 16:

 17:  }

0011 81        RTS
```

21.3 Using Variables in EEPROM

The C language does not explicitly support writing variables to EEPROM, however, you may occasionally need to perform such activities. The processor-specific examples in this section can be easily adapted for other processors.

The topics covered here are as follows:

- Linker Parameter File
- The Application

21.3.1 Linker Parameter File

Define your RAM or ROM areas in your linker parameter file (refer to the following listing). However, to avoid initializing the memory range during normal startup, declare the EEROM memory as NO_INIT.

Listing: Linker Parameter File

```
LINK test.abs
NAMES test.o startup.o ansi.lib END

SECTIONS

  MY_RAM = READ_WRITE 0x800 TO 0x801;

  MY_ROM = READ_ONLY  0x810 TO 0xAFF;

  MY_STK = READ_WRITE 0xB00 TO 0xBFF;

  EEPROM =
NO_INIT    0xD00 TO 0xD01;

PLACEMENT

  DEFAULT_ROM  INTO  MY_ROM;

  DEFAULT_RAM  INTO  MY_RAM;

  SSTACK       INTO  MY_STK;

  EEPROM_DATA  INTO  EEPROM;

END

/* set reset vector to the _Startup function defined in startup code */

VECTOR ADDRESS 0xFFFE _Startup
```

21.3.2 The Application

The example in the following erases or writes an EEPROM word. Consult the technical documentation for your derivative or CPU to adapt the example for your processor.

NOTE

Consult your technical documentation before initiating write operations to the EEPROM. Limits on the number of writes available vary depending on the device and manufacturer.

Listing: Erasing and Writing an EEPROM

```
/*
 Definition of a variable in EEPROM

   The variable VAR is located in EEPROM.

   - It is defined in a user-defined segment EEPROM_DATA

   - In the PRM file, EEPROM_DATA is placed at address 0xD00.

Be careful, the EEPROM can only be written a limited number of times.

Running this application too frequently may surpass this limit and the
EEPROM may be unusable afterwards.

*/

#include <hidef.h>

#include <stdio.h>

#include <math.h>

/* INIT register. */

typedef struct {

  union {

    struct {

      unsigned int   bit0:1;

      unsigned int   bit1:1;

      unsigned int   bit2:1;

      unsigned int   bit3:1;

      unsigned int   bit4:1;

      unsigned int   bit5:1;

      unsigned int   bit6:1;

      unsigned int   bit7:1;

    } INITEE_Bits;

    unsigned char INITEE_Byte;

  } INITEE;

} INIT;

volatile INIT INITEE @0x0012;

#define EEON INITEE.INITEE.INITEE_Bits.bit0
```

```
/* EEPROG register. */
volatile struct {
  unsigned int   EEPGM:1;
  unsigned int   EELAT:1;
  unsigned int   ERASE:1;
  unsigned int   ROW:1;
  unsigned int   BYTE:1;
  unsigned int   dummy1:1;
  unsigned int   dummy2:1;
  unsigned int   BULKP:1;
} EEPROG @0x00F3;
/* EEPROT register. */
volatile struct {
  unsigned int   BPROT0:1;
  unsigned int   BPROT1:1;
  unsigned int   BPROT2:1;
  unsigned int   BPROT3:1;
  unsigned int   BPROT4:1;
  unsigned int   dummy1:1;
  unsigned int   dummy2:1;
  unsigned int   dummy3:1;
} EEPROT @0x00F1;
#pragma DATA_SEG EEPROM_DATA
unsigned int VAR;
#pragma DATA_SEG DEFAULT
void EraseEEPROM(void) {
  /* Function used to erase one word in the EEPROM. */
  unsigned long int i;
  EEPROG.BYTE = 1;
  EEPROG.ERASE = 1;
  EEPROG.EELAT = 1;
  VAR = 0;
  EEPROG.EEPGM =1;
  for (i = 0; i<4000; i++) {
    /* Wait until EEPROM is erased. */
```

```
  }
  EEPROG.EEPGM = 0;
  EEPROG.EELAT = 0;
  EEPROG.ERASE = 0;
}
void WriteEEPROM(unsigned int val) {
  /* Function used to write one word in the EEPROM. */
  unsigned long int i;
  EraseEEPROM();
  EEPROG.ERASE = 0;
  EEPROG.EELAT = 1;
  VAR = val;
  EEPROG.EEPGM = 1;
  for (i = 0; i<4000; i++) {
  /* Wait until EEPROM is written. */
  }
  EEPROG.EEPGM = 0;
  EEPROG.EELAT = 0;
  EEPROG.ERASE = 0;
}
void func1(void) {
  unsigned int i;
  unsigned long int ll;
  i = 0;
    do
    {
    i++;
    WriteEEPROM(i);
    for (ll = 0; ll<200000; ll++) {
    }
  }
  while (1);
}
void main(void) {
  EEPROT.BPROT4 = 0;
```

```
  EEON=1;
  WriteEEPROM(0);
  func1();
}
```

21.4 General Optimization Hints

Use these hints to reduce the size of your application:

- Verify that you need the full startup code:
  - If you have no initialized data, you can ignore or remove the copy-down
  - If you do not need initialized memory, you can remove the zero-out.
  - If you do not need either, completely remove the startup code and set up your stack directly in the main routine. Use `INITmain` in the `prm` file as the startup or entry into your main application routine.
- Check the compiler options:
  - -OdocF: Dynamic Option Configuration for Functions compiler option increases compilation speed, but decreases code size. Try `-OdocF=-or`.
  - -Li: List of Included Files to ".inc" File writes a log file that displays the statistics for each single option.
- Find out if you can use IEEE32 for both float and double. (See -T: Flexible Type Management for configuration. Remember to link the corresponding ANSI-C library.)
- Use smaller data types whenever possible (for example, 16 bits instead of 32 bits).
- Check runtime routines in the map file (runtime routines usually have a _ prefix).
  - Check for 32-bit integral routines (for example, _LADD).
  - Check if you need the long arithmetic.
- Check the size of enumerations. `enums` have the size `int` by default. If possible, set `enums` to an unsigned 8-bit (see option -T, or use -TE1uE).
- Find out if you are using switch tables (look in the map file as well). Configure switch table use speed or code density (see -CswMinSLB: Minimum Number of Labels for Switch Search Tables).
- Use the linker to overlap ROM areas when reasonable (see the `-COCC` option).

21.5 Executing Application from RAM

For performance reasons, you can copy an application from ROM to RAM and execute it from RAM. The following procedure outlines this process.

1. Link your application with code located in RAM.
2. Generate an S-Record File.
3. Modify the startup code to copy the application code.
4. Link the application with the S-Record File previously generated.

NOTE

It is recommended that you generate a ROM library for your application. This allows you to debug your final application easily (including the copying of the code).

The following sections describe each step:

- ROM Library Startup File
- Generate an S-Record File
- Modify the Startup Code
- Application PRM File
- Copying Code from ROM to RAM

21.5.1 ROM Library Startup File

A ROM Library requires a very simple startup file, containing only the definition from the startup structure. Usually a ROM library startup file looks as follows:

```
#include "startup.h"


/* read-only: _startupData is allocated in ROM and ROM
Library PRM File */


struct _tagStartup _startupData;
```

You must generate a PRM file to set code placement in RAM. Because the compiler generates absolute code, the linker must know the final location of the code to generate correct code for the function call.

In the PRM file, you must:

- Specify the name of the application entry points in the ENTRIES block

- Specify the application's main function
- Specify the function associated with an interrupt vector

The following listing shows a PRM file that copies and executes code at address 0x7000.

Listing: Linker Parameter File

```
LINK fiboram.abs AS ROM_LIB
NAMES  myFibo.o start.o

END

SECTIONS

  MY_RAM = READ_WRITE 0x4000 TO 0x43FF;

  MY_ROM = READ_ONLY  0x7000 TO 0xBFFF; /* Dest. Address in RAM area */

PLACEMENT

  DEFAULT_ROM, ROM_VAR, STRINGS  INTO  MY_ROM;

  DEFAULT_RAM                    INTO   MY_RAM;

END

ENTRIES

  myMain

END
```

NOTE

You cannot use a main function in a ROM library. Use another name for the application's entry point. In the example above, we used `myMain`.

21.5.2 Generate an S-Record File

Use the Burner utility to generate the required S-Record file. The Burner utility generates the file when you click **Execute** in the burner dialog.

NOTE

Initialize the field `From` with zero and the field `Length` with a value greater than the last byte used for the code. If byte `0xFFFF` is used, then `Length` must be at least `10000`.

21.5.3 Modify the Startup Code

Modify the final application startup code to include code that copies the code from RAM to ROM. You must explicitly call the application's entry point (the entry point is located in the ROM library).

21.5.4 Application PRM File

Use an offset to link the S-Record File (generated previously) to the application.

The application's PRM file places the application code at address 0x800 in ROM and copies the code to address 0x7000 in RAM. The following listing shows the PRM file.

Listing: Linker Parameter File

```
LINK fiborom.abs
NAMES  mystart.o fiboram.abs ansis.lib END

SECTIONS

    MY_RAM = READ_WRITE 0x5000 TO 0x53FF;

    MY_ROM = READ_ONLY  0x0600 TO 0x07FF;

PLACEMENT

    DEFAULT_ROM, ROM_VAR, STRINGS  INTO MY_ROM;

    DEFAULT_RAM                    INTO MY_RAM;

END

STACKSIZE 0x100

VECTOR 0 _Startup /* set reset vector on startup function */

HEXFILE fiboram.s1 OFFSET 0xFFFF9800 /* 0x800 - 0x7000 */
```

NOTE

The Compiler adds the offset specified in the HEXFILE command to each record in the S-Record File, and encodes the code at address 0x700 to address 0x800.

If using CodeWarrior tools, then the CodeWarrior IDE passes all the names in the NAMES...END directive directly to the linker. Ensure that the NAMES...END directive is empty.

21.5.5 Copying Code from ROM to RAM

You must implement a function that copies the code from ROM to RAM.

To place the application code at address 0x800 in ROM and copy it to address 0x7000 in RAM, implement a copy function (refer to the following listing).

Listing: Definition of the CopyCode() Function

```
/* Start address of the application code in ROM. */
#define CODE_SRC  0x800

/* Destination address of the application code in RAM. */

#define CODE_DEST 0x7000

#define CODE_SIZE 0x90 /* Size of the code which must be copied.*/

void CopyCode(void) {

  unsigned char *ptrSrc, *ptrDest;

  ptrSrc = (unsigned char *)CODE_SRC;

  ptrDest = (unsigned char *)CODE_DEST;

  memcpy (ptrDest, ptrSrc, CODE_SIZE);
}
```

The following topics are covered in this section:

- Invoking Application's Entry Point in Startup Function
- Defining Dummy Main Function

21.5.5.1 Invoking Application's Entry Point in Startup Function

The startup code must call the application's entry point, located in the ROM library. You must explicitly call this function by name. Ideally, call the application's entry point just before calling the application's main routine (refer the following listing).

Listing: Invoking the Application's Entry Point

```
void _Startup(void) {
  ... set up stack pointer ...

  ... zero out ...

  ... copy down ...

  CopyCode();
```

```
  ... call main ...
}
```

21.5.5.2 Defining Dummy Main Function

The ROM library contains the main function. Define a dummy main function in the startup module (refer the following listing).

Listing: Definition of a Dummy Main Function

```
#pragma NO_ENTRY
#pragma NO_EXIT

void main(void) {

  asm NOP;

}
```

21.6 Frequently Asked Questions (FAQs) and Troubleshooting

This section provides some tips to solve the most commonly encountered problems, covered in the following topic:

- Making Applications

21.6.1 Making Applications

If the compiler or linker crashes, isolate the construct causing the crash and send a bug report to Freescale support. Other common problems are:

21.6.1.1 The Compiler Reports an Error, but WinEdit Does not Display it.

This means that WinEdit did not find the EDOUT file (that is, the compiler wrote it to a place not expected by WinEdit). Some steps to take include:

- Ensure that the DEFAULTDIR: Default Current Directory environment variable is not set and that the project directory is set correctly.
- In WinEdit 2.1, make sure that the OUTPUT entry in the file WINEDIT.INI is empty.

21.6.1.2 Some Programs Cannot Find a File.

Some steps to take include:

- Make sure the environment is set up correctly.
- Check WinEdit's project directory.

For more information read the Input Files section of the Files chapter.

21.6.1.3 The Compiler Seems to Generate Incorrect Code.

First, determine whether the code is really incorrect. Sometimes the operator-precedence rules of ANSI-C do not give the results one would expect, and sometimes faulty code appears correct. Consider the following example.

Listing: Possibly faulty code?

```
if (x & y != 0) ...
evaluates as:

if (x & (y != 0)) ...

but not as:

if ((x & y) != 0) ...
```

C's integral promotion rules sometimes produce unexpected behavior. Characters are usually (sign-)extended to integers. This sometimes has unexpected results (refer the following listing).

Listing: If Condition is Always FALSE

```
unsigned char a, b;
b = -8;

a = ~b;
```

```
if (a == ~b) ...
```

The if condition in this example is always false, because extending `a` results in `0x0007`, while extending `b` gives `0x00F8` and the `~` results in `0xFF07`. If the code contains a bug, isolate the construct causing it and send a bug report to Freescale support.

21.6.1.4 The code seems to be correct, but the application does not work.

Ensure that the hardware is set up correctly (for example, using chip selects). Some memory expansions require a special access mode (for example, only word accesses). If memory is accessible only in a certain way, use inline assembly or use the `volatile` keyword.

21.6.1.5 The linker cannot handle an object file.

Usually this is caused by compiling object files with different compiler versions, or having different flag settings. Make sure that all object files are compiled with the latest version of the compiler and with the same flags for memory models and floating point formats. If not, recompile them.

21.6.1.6 The make Utility does not Make the entire Application.

Probably you did not specify the target on the command line. In this case, the make utility assumes the target of the first rule is the top target. Either put the rule for your application as the first in the make file, or specify the target on the command line.

21.6.1.7 The make utility unnecessarily re-compiles a file.

This problem appears when you have short source files in your application. MS-DOS only saves the time of last modification of a file with an accuracy of ±2 seconds. When the compiler compiles two files in that time, both have the same time stamp. The make utility makes the safe assumption that if one file depends on another file with the same time stamp, the first file must be recompiled. There is no way to solve this problem.

21.6.1.8 The help file cannot be opened by double clicking on it in the File Manager or in the Explorer.

The compiler help file is a true Win32 help file. To open the compiler help file, use `Winhlp32.exe`. The `winhlp32.exe` program resides either in the Windows directory (usually `C:\windows`, `C:\win95` or `C:\winnt`) or in its system (`Win32s`) or `system32` (Windows 2000, Windows XP, or Windows Vista operating systems) subdirectory. The Win32s distribution also contains `Winhlp32.exe`.

To change the association with Windows 95 or Windows NT either (1) use the explorer menu **View > Options** and then the **File Types** tab or (2) select any help file and press the *Shift* key. Hold it while opening the context menu by clicking on the right mouse button. Select **Open with** from the menu. Enable the **Always using this program** checkbox and use the **Other** option to select the `winhlp32.exe` file.

21.6.1.9 How can I allocate constant objects in ROM?

Use #pragma INTO_ROM: Put Next Variable Definition into ROM and the -Cc: Allocate Const Objects into ROM compiler option.

21.6.1.10 The compiler cannot find my source file. What is wrong?

In the `default.env` file, ensure that the source file path is set in the GENPATH: #include "File" Path environment variable. In addition, use the -I: Include File Path compiler option to specify the include file path.

21.6.1.11 How can I switch off smart linking?

Add + after the object in the `NAMES` list of the `prm` file.

When using CodeWarrior tools and the ELF/DWARF object-file format (see -F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format) compiler option, you can link everything in the object within an `ENTRIES...END` directive in the linker prm file:

```
ENTRIES fibo.o:* END
```

This is NOT supported in the HIWARE object-file format.

21.6.1.12 How can I avoid the `no access to memory' warning?

In the simulator or debugger, change the memory configuration mode (menu **Simulator > Configure**) to **auto on access** .

21.6.1.13 How can I load the same memory configuration every time the simulator or debugger starts?

Save the memory configuration under `default.mem`. For example, select **Simulator > Configure > Save** and enter `default.mem`.

21.6.1.14 How can I automatically start a loaded program in the simulator or debugger and stop at a specified breakpoint?

Define the `postload.cmd` file. For example:

```
bs &main t


g
```

21.6.1.15 How can I produce an overview of all the compiler options?

Type in -H: Short Help on the command line of the compiler.

21.6.1.16 How can I call a custom startup function after reset?

In the PRM file, use:

```
INIT myStartup
```

21.6.1.17 How can I use a custom name for the main() function?

In the PRM file, use:

```
MAIN myMain
```

21.6.1.18 How can I set the reset vector to the beginning of the startup code?

Use this line in the prm file:

```
/* set reset vector on _Startup */


VECTOR ADDRESS 0xFFFE _Startup
```

21.6.1.19 How can I configure the compiler for the editor?

Open the compiler, select **File > Configuration** from the menu, and choose **Editor Settings** .

21.6.1.20 Where are configuration settings saved?

In the `project.ini` file.

21.6.1.21 What should be done when "error while adding default.env options" appears after starting the compiler?

Change the Compiler options to those in the `default.env` file. Save them in the `project.ini` file by clicking the **Save** button in the compiler.

21.6.1.22 After starting the ICD Debugger, an "Illegal breakpoint detected" error appears. Why?

The cable might be too long. The maximum length for unshielded cables is about 20 cm. This error may also indicate too much electrical noise in the environment.

21.6.1.23 How can I write initialized data into the ROM area?

Use the `const` qualifier, and compile the source with the -Cc: Allocate Const Objects into ROM option.

21.6.1.24 There are communication problems or it loses communication.

The cable might be too long. The maximum length for unshielded cables is about 20 cm. This error may also indicate too much electrical noise in the environment.

21.6.1.25 What should be done if an assertion happens (internal error)?

Extract the source where the assertion appears and send it as a zipped file with all the headers, options and versions of all tools.

21.6.1.26 How can I get help on an error message?

Click on the message and press F1 to start up the help file. You can also copy the message number, open the PDF manual, and search on the message number.

21.6.1.27 How can I get help on an option?

Open the compiler and type -H: Short Help into the command line. A list of all options appears with a short description of each. You can also look in the manual for detailed information. A third way is to select the option and press F1 in the **Options Settings** panel.

21.6.1.28 I cannot connect to my target board using an ICD Target Interface.

Check the following:

- Is the parallel port working correctly? Try to print a document using the parallel port. If successful, this ensures that the parallel port is available and connected.
- Is the BDM connector designed according to the specification from P&E?
- If you are running a Windows NT or Win98 operating system, you need to install an additional driver in order to be able to communicate with the software. (See NT Installation Notice in the debugger ICD Target Interface Manual.)
- Do not extend the original ICD Cable from P&E. Extending this cable often generates communication problems. Maximum cable length is 25 cm.
- The PC may be too fast for the ICD cable. You can slow down the communication between the PC and the Target using the environment variable BMDELAY (for example, BMDELAY=50).

21.7 Bug Reports

If you cannot solve your problem, you may need to contact our Technical Support Department. Isolate the problem; if it is a compiler problem, write a short program reproducing the problem. Then send or fax your bug report to your local distributor, who will forward it to the Technical Support Department.

The report type informs us of the urgency level of the bug report. The classifications are:

- Information

  This describes things you would like to see improved in a future major release.

- Bug

  This is an error which requires a workaround for you to continue work. Frequently we are able to supply workaround solutions for bugs. (If you already have a workaround, we'd like to know about it, too!) Bugs will be fixed in the next release.

- Critical Bug

  A grave error that makes it impossible for you to continue with your work.

21.8 EBNF Notation

This chapter gives a short overview of the Extended Backus-Naur Form (EBNF) notation, which is frequently used in this document to describe file formats and syntax rules.

Listing: EBNF Syntax

```
   ProcDecl   = PROCEDURE
 ( ArgList ).
   ArgList    = Expression {, Expression}.

   Expression = Term (*|/) Term.

   Term       = Factor AddOp Factor.

   AddOp      = +|-.

   Factor     = ([-] Number)|( Expression ).
```

The EBNF notation is used to express the syntax of context-free languages. The EBNF grammar consists of a rule set called *productions* of the form:

```
LeftHandSide = RightHandSide.
```

The left-hand side is a non-terminal symbol. The right-hand side describes the composition.

EBNF consists of the following symbols. A brief discussion of each symbol follows.

- Terminal Symbols
- Non-Terminal Symbols
- Vertical Bar

- Brackets
- Parentheses
- Production End
- EBNF Syntax
- Extensions

21.8.1 Terminal Symbols

Terminal symbols (or terminals) are the basic symbols which form the language described. In the above example, PROCEDURE is a terminal.

21.8.2 Non-Terminal Symbols

Non-terminal symbols (non-terminals) are syntactic variables and have to be defined in a production, that is, they have to appear on the left hand side of a production somewhere. In the example above, there are many non-terminals, for example, `ArgList` or `AddOp`.

21.8.3 Vertical Bar

The vertical bar `"|"` denotes an alternative, that is, either the left or the right side of the bar can appear in the language described, but one of them must appear (for example, the third production above means "an expression is a term followed by either a `"*"` or a `"/"` followed by another term").

21.8.4 Brackets

Parts of an EBNF production enclosed by `"["` and `"]"` are optional. They may appear exactly once in the language, or they may be skipped. The minus sign in the last production above is optional, both `-7` and `7` are allowed.

The repetition is another useful construct. Any part of a production enclosed by `"{"` and `"}"` may appear any number of times in the language described (including zero, that is, it may also be skipped). `ArgList` above is an example: an argument list is a single expression or a list of any number of expressions separated by commas.

NOTE

The syntax in the example does not allow empty argument lists.

21.8.5 Parentheses

For better readability, normal parentheses may be used for grouping EBNF expressions, as is done in the last production of the example. Note the difference between the first and the second left bracket. The first one is part of the EBNF notation. The second one is a terminal symbol and may appear in the language.

21.8.6 Production End

A production is always terminated by a period.

21.8.7 EBNF Syntax

The definition of EBNF in the EBNF language is:

Listing: EBNF Definition in EBNF language

```
Production  = NonTerminal = Expression ..
Expression  = Term {| Term}.

Term        = Factor {Factor}.

Factor      = NonTerminal

              | Terminal

              | "(" Expression ")"

              | "[" Expression "]"

              | "{" Expression "}".

Terminal    = Identifier | """ <any char> """.
```

```
NonTerminal  = Identifier.
```

The identifier for a non-terminal can be any name you like. Terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.

21.8.8 Extensions

In addition to this standard definition of EBNF, the following notational conventions are used.

The counting repetition: Anything enclosed by `{` and `}` and followed by a superscripted expression `x` must appear exactly `x` times. `x` may also be a non-terminal. In the following example, exactly four stars are allowed:

```
Stars = {*}4.
```

The size in bytes: Any identifier immediately followed by a number *n* in square brackets (`[` and `]`) may be assumed to be a binary number with the most significant byte stored first, having exactly *n* bytes. The following listing shows the example.

Listing: Example of 4-Byte Identifier FilePos

```
Struct = RefNo FilePos[4].
```

In some examples, text is enclosed by `<` and `>`. This text is a meta-literal, that is, whatever the text says may be inserted in place of the text (confer `<any char>` in the above listing, where any character can be inserted).

21.9 Abbreviations and Lexical Conventions

The following table lists some programming terms used in this manual.

Table 21-1. Common Terminology

Topic	Description
ANSI	American National Standards Institute
Compilation Unit	Source file to be compiled, includes all included header files
Floating Type	Numerical type with a fractional part, for example, `float`, `double`, `longdouble`

Table continues on the next page...

Table 21-1. Common Terminology (continued)

Topic	Description
HLI	High-level Inline Assembly
Integral Type	Numerical type without a fractional part, for example, `char`, `short`, `int`, `long`, `longlong`

21.10 Number Formats

Valid constant floating number suffixes are `f` and `F` for float and `l` or `L` for long double. Note that floating constants without suffixes are double constants in ANSI. For exponential numbers use `e` or `E`. Use `-` and `+` for signed representation of the floating number or the exponent.

The following table lists the suffixes that are supported.

Table 21-2. Supported Number Suffixes

Constant	Suffix	Type
floating	`F`	`float`
floating	`L`	`long double`
integral	`U`	`unsigned in t`
integral	`uL`	`unsigned long`

Suffixes are not case-sensitive, for example, `ul`, `Ul`, `uL` and `UL` all denote an `unsigned long` type. The following listing shows examples of these numerical formats.

Listing: Examples of Supported Number Suffixes

```
+3.15f  /* float */
-0.125f /* float */

3.125f  /* float */

0.787F  /* float */

7.125   /* double */

3.E7    /* double */

8.E+7   /* double */

9.E-7   /* double */

3.2l    /* long double */

3.2e12L /* long double */
```

21.11 Precedence and Associativity of Operators for ANSI-C

The following table gives an overview of the precedence and associativity of operators.

Table 21-3. ANSI-C Precedence and Associativity of Operators

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`? :`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

NOTE

Unary +, - and * have higher precedence than the binary forms.

ANSI-C syntax determines precedence and associativity (ANSI/ISO 9899-1990, p. 38 and Kernighan/ Ritchie, "*The C Programming Language*", Second Edition).

Listing: Examples of Operator Precedence and Associativity

```
  if (a == b&&c) and
  if ((a == b)&&c) are equivalent.

However,

  if (a == b|c)

is the same as

  if ((a == b)|c)

  a = b + c * d;
```

In the above listing, the operator-precedence adds the product of (`c*d`) to `b`, and then assigns that sum to `a`.

In the following listing, the associativity rule first evaluates `c+=1`, then assigns `b` to the value of `b` plus `(c+=1)`, and then assigns the result to `a`.

Listing: Three Assignments in One Statement

```
a = b += c += 1;
```

21.12 List of all Escape Sequences

List of all Escape Sequences lists all escape sequences. Use these escape sequences inside strings, if needed (for example, for `printf`).

Table 21-4. Escape Sequences

Description	Escape Sequence
Line Feed	`\n`
Tabulator sign	`\t`
Vertical Tabulator	`\v`
Backspace	`\b`
Carriage Return	`\r`
Line feed	`\f`
Bell	`\a`
Backslash	`\\`
Question Mark	`\?`
Quotation Mark	`\^`
Double Quotation Mark	`\"`
Octal Number	`\ooo`
Hexadecimal Number	`\xhh`

Chapter 22
Global Configuration File Entries

This appendix describes the possible entries in the `mcutools.ini` global configuration file, and contains these sections:

- [Options] Section
- [XXX_Compiler] Section
- [Editor] Section
- Example

22.1 [Options] Section

This section documents the following possible entries in the `[Options]` section of the file `mcutools.ini`:

- DefaultDir

22.1.1 DefaultDir

Arguments

Default directory to be used.

Description

Specifies the current directory for all tools on a global level (see also environment variable DEFAULTDIR: Default Current Directory).

Example

```
DefaultDir=C:\install\project
```

22.2 [XXX_Compiler] Section

This section documents the following possible entries in an [*XXX*_Compiler] section of the mcutools.ini file:

NOTE

XXX is a placeholder for the name of the actual backend. For example, for the HC08 compiler, the name of this section is [HC08_Compiler].

- SaveOnExit
- SaveAppearance
- SaveEditor
- SaveOptions
- RecentProject0, RecentProject1
- TipFilePos
- ShowTipOfDay
- TipTimeStamp

22.2.1 SaveOnExit

Arguments

1/0

Description

Set to 1 to store the configuration when the compiler is closed. Clear to 0 if storing is unnecessary. The compiler does not ask to store a configuration in either case.

22.2.2 SaveAppearance

Arguments

1/0

Description

Set to 1 to store the visible topics when writing a project file. Clear to 0 otherwise. The command line, its history, the window position, and other topics belong to this entry.

22.2.3 SaveEditor

Arguments

1/0

Description

Set to 1 to store the visible topics when writing a project file. Clear to 0 otherwise. The editor setting contains all information of the **Editor Configuration** dialog.

22.2.4 SaveOptions

Arguments

1/0

Description

Set to 1 to save the options when writing a project file. Clear to 0 to exit without saving. The option also contains the message settings.

22.2.5 RecentProject0, RecentProject1

Arguments

Names of the last and prior project files

Description

The Compiler updates this list when a project loads or saves. Its current content shows in the file menu.

Example

```
SaveOnExit=1

SaveAppearance=1

SaveEditor=1

SaveOptions=1

RecentProject0=C:\myprj\project.ini

RecentProject1=C:\otherprj\project.ini
```

22.2.6 TipFilePos

Arguments

Any integer, for example, 236

Description

Actual position in tip of the day file. Used to show different tips at different calls.

Saved

Always saved when saving a configuration file.

22.2.7 ShowTipOfDay

Arguments

```
0/1
```

Description

Used to enable showing the **Tip of the Day** dialog at startup. Set to 1 to enable the **Tip of the Day** . Clear to 0 to open **Tip of the Day** through the **Help** menu only.

Saved

Always saved when saving a configuration file.

22.2.8 TipTimeStamp

Arguments

date and time

Description

Date and time when the tips were last used.

Saved

Always saved when saving a configuration file.

22.3 [Editor] Section

This section documents the following possible entries in the `[Editor]` section of the `mcutools.ini` file:

- Editor_Name
- Editor_Exe
- Editor_Opts
- Example [Editor] Section

22.3.1 Editor_Name

Arguments

The name of the global editor

Description

Specifies the name which is displayed for the global editor. This entry has only a descriptive effect. Its content is not used to start the editor.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

22.3.2 Editor_Exe

Arguments

The name of the executable file of the global editor

Description

Specifies the filename that is called (for showing a text file) when the global editor setting is active. In the **Editor Configuration** dialog, the global editor selection is active only when this entry is present and not empty.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

22.3.3 Editor_Opts

Arguments

The options for use with the global editor

Description

Specifies options used for 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 this entry.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

22.3.4 Example [Editor] Section

```
[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f
```

22.4 Example

The following listing shows a typical `mcutools.ini` file.

Listing: Typical mcutools.ini File Layout

```
[Installation]
Path=c:\Freescale

Group=ANSI-C Compiler

[Editor]

editor_name=notepad

editor_exe=C:\windows\notepad.exe

editor_opts=%f

[Options]

DefaultDir=c:\myprj

[XXXX_Compiler]

SaveOnExit=1

SaveAppearance=1

SaveEditor=1

SaveOptions=1

RecentProject0=c:\myprj\project.ini

RecentProject1=c:\otherprj\project.ini

TipFilePos=0

ShowTipOfDay=1

TipTimeStamp=Jan 21 2009 17:25:16
```

Chapter 23
Local Configuration File Entries

This appendix describes the possible 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. This appendix contains these sections:

23.1 [Editor] Section

This section documents the possible entries in an `[`*Editor*`]` section of a *project*`.ini` file.

23.1.1 Editor_Name

Arguments

The name of the local editor

Description

Specifies the name displayed for the local editor. This entry has only a descriptive effect. Its content is not used to start the editor.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog. This entry has the same format as the global **Editor Configuration** in the `mcutools.ini` file.

23.1.2 Editor_Exe

Arguments

The name of the executable file of the local editor

Description

Specifies the filename that is used for a text file when the local editor setting is active. In the **Editor Configuration** dialog, the local editor selection is active only when this entry is present and not empty.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog. This entry has the same format as the global **Editor Configuration** in the `mcutools.ini` file.

23.1.3 Editor_Opts

Arguments

Local editor options

Description

Specifies local editor options. If this entry is absent 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 this entry.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog. This entry has the same format as the global **Editor Configuration** in the `mcutools.ini` file.

23.1.4 Example [Editor] Section

```
[Editor]


editor_name=notepad


editor_exe=C:\windows\notepad.exe


editor_opts=%f
```

23.2 [XXX_Compiler] Section

This section documents the following possible entries in an [*XXX*_Compiler] section of a *project*.ini file:

NOTE

XXX is a placeholder for the name of the actual backend. For example, for the HC08 compiler, the name of this section is [HC08_Compiler].

- RecentCommandLineX
- CurrentCommandLine
- StatusbarEnabled
- ToolbarEnabled
- WindowPos
- WindowFont
- Options
- EditorType
- EditorCommandLine
- EditorDDEClientName
- EditorDDETopicName
- EditorDDEServiceName

23.2.1 RecentCommandLineX

NOTE

X is a placeholder for an integer.

Arguments

String with a command line history entry, for example, `fibo.c`

Description

This list of entries contains the content of the command line history.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

23.2.2 CurrentCommandLine

Arguments

String with the command line, for example, `fibo.c -w1`

Description

The currently visible command line content.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

23.2.3 StatusbarEnabled

Arguments

1/0

Special

This entry is considered only at startup. Later load operations do not use it.

Description

Use to enable or disable the status bar. Set to 1 to make the status bar visible. Clear to 0 to hide the status bar.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

23.2.4 ToolbarEnabled

Arguments

1/0

Special

This entry is considered only at startup. Later load operations do not use it.

Description

Use to enable or disable the toolbar. Set to 1 to make the toolbar visible. Clear to 0 to hide the toolbar.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

23.2.5 WindowPos

Arguments

10 integers, for example, " `0,1,-1,-1,-1,-1,390,107,1103,643`"

Special

This entry is considered only at startup. Later load operations do not use it.

Changes of this entry do not show the "*" in the title.

Description

This number contains the position and the state of the window (maximized) and other flags.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

23.2.6 WindowFont

Arguments

```
size: == 0 -> generic size, < 0 -> font character height,

                            > 0 -> font cell height


weight: 400 = normal, 700 = bold (valid values are 0 -
1000)


italic: 0 == no, 1 == yes


font name: max 32 characters
```

Description

Font attributes.

Saved

Only with **Appearance** set in the **File > Configuration > Save Configuration** dialog.

Example

```
WindowFont=-16,500,0,Courier
```

23.2.7 Options

Arguments

-W2

Description

The currently active option string. This entry also stores messages and is quite long.

Saved

Only with **Options** set in the **File > Configuration > Save Configuration** dialog.

23.2.8 EditorType

Arguments

0/1/2/3

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

For details see Editor Settings Dialog Box.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

23.2.9 EditorCommandLine

Arguments

Command line for the editor.

Description

Command line content to open a file. For details see Editor Settings Dialog Box.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

23.2.10 EditorDDEClientName

Arguments

Client command, for example, " `[open(%f)]`"

Description

Name of the client for DDE Editor Configuration. For details see Editor Started with DDE.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

23.2.11 EditorDDETopicName

Arguments

Topic name. For example, "system"

Description

Name of the topic for DDE Editor Configuration. For details, see Editor Started with DDE

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

23.2.12 EditorDDEServiceName

Arguments

Service name. For example, "system"

Description

Name of the service for DDE Editor Configuration. For details, see Editor Started with DDE.

Saved

Only with **Editor Configuration** set in the **File > Configuration > Save Configuration** dialog.

23.3 Example

The following listing shows a typical configuration file layout (usually *project*`.ini`).

Listing: Typical Local Configuration File Layout

```
[Editor]
Editor_Name=notepad

Editor_Exe=C:\windows\notepad.exe

Editor_Opts=%f

[XXX_Compiler]

StatusbarEnabled=1

ToolbarEnabled=1

WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643

WindowFont=-16,500,0,Courier

Options=-w1

EditorType=3

RecentCommandLine0=fibo.c -w2

RecentCommandLine1=fibo.c

CurrentCommandLine=fibo.c -w2

EditorDDEClientName=[open(%f)]

EditorDDETopicName=system

EditorDDEServiceName=msdev

EditorCommandLine=C:\windows\notepad.exe %f
```

Chapter 24
Known C++ Issues in the HC(S)08 Compilers

This appendix describes the known issues when using C++ with the HC(S)08 compilers, and contains these sections:

24.1 Template Issues

This section describes unsupported template features.

- Template specialization is unsupported. Example:

```
template <class T> class C {};
template <> class C<double> {};
---------^-------------------- ERROR
```

- Declaring a template in a class is unsupported. Example:

```
struct S {
       template <class T1, class T2> void f(T1, T2) {}
```

```
};
-     template <class T> struct S<...>
-template <int i>
```

- Non-template parameters are unsupported. Example:

```
    template<> int f()
-  S03< ::T03[3]> s03;
--------------^------------------Doesn't know global scope ::
   template <int i, class P> struct S {}
    S<0xa301, int(*)[4][3]> s0;
-------------------------^--------Wrong type of template argument
```

- Implicit instantiations are unsupported. Example:

```
template <int i > struct A{
         A<i>() {}
-----------------^----------------ERROR implicit instantiation
      }
-     void g00(void) {}
                void g00(U) {}
      int  g00(char) { return 0; }
------^------------------------ERROR: Function differ in return type
```

- Accepting a `template` template parameter is unsupported. Example:

```
template <template <class P> class X, class T> struct A{}
```

- Defining a static function template is unsupported. Example:

```
   template <class T> static int f(T t) {return 1}
----------------^--ERROR : Illegal storage class
```

24.2 Operators

This section describes operator-related limitations and issues as well as unsupported operator features.

- Relational o perators other than `==' are unsupported for function pointers.
- Operators in expressions are unsupported. Example:

```
- struct A { };
  void operator*(A) { counter++; }
  enum B{ };
  int operator*(B) { return 0; }
  -------------------^-----Function differs in return type only
                           (found 'void ' expected 'int ')
- struct A{
     operator int*(){return &global;}
  }
  A a;
  (void)*a;
  --------^---------------Compile ERROR
- struct A{};
     struct B:struct A{};
     int operator*(A) {return 1;}
     int f() {
     B b;
     return (*b);
   -----------------^----------------Illegal cast operation
     }
- int operator->*(B,int){ return 1; }
  ----------------^------ERROR: unary operator must have one parameter
```

- When an expression uses an operator, a member function with the operator's name should not hide a non-member function with the same name. Example:

```
struct A {
       void operator*() { }
       void test();
};
void operator*(S, int) { } // not hidden by S::operator*()
void S::test(){
      S s;
      (void) (s * 3);
>
   --------------^------------------Compile ERROR
```

- Explicit operator calls are unsupported. Example:

```
        struct B {
              operator int() { return 1; }
        };
        B b;
        b.operator int();
------------^------------ERROR: Not supported explicit operator call
```

The other operator-related limitations and issues as well as unsupported operator features for the following operators are described here:

- Binary Operators
- Unary operators
- Equality Operators

24.2.1 Binary Operators

The following binary operator functions are unsupported:

- Implementing the binary `->*` operator as a non-member function with two parameters. Example:

```
friend long operator->* (base x, base y) ;
```

- Implementing the binary `->*` operator as a non-static member function with one parameter. Example:

```
int operator ->* (C) ;
```

- Overloaded operators are unsupported. Example:

```
struct S {
      int m;
      template <class T> void operator+=(T t) { m += t; } //
ERROR at template
};
```

24.2.2 Unary operators

The following unary operator functions are unsupported:

- Implementing the unary ~ operator as a non-member function with one parameter. Example:

```
   int operator ~(C &X) { return 1; }
   int tilda (C &X)     { return 1; }
   if (~c != tilda(c))
----------^----------------------------ERROR: Integer-operand expected
```

- Implementing the unary ! operator as a non-member function with one parameter. Example:

```
  class A{};
  int operator!(A &X) { return 1; }
  int bang_(A &X) { return 1; }
  A a;
  if ((!a) != (bang_(a)))
---------^--------ERROR : Arithmetic type or pointer expected
```

- Logical OR operators are unsupported. Example:

```
class X {
public:
      operator int() {i = 1; return 1;}
} x;
(void) (0 || x);
-----------^------------ERROR
```

- Conditional operators are unsupported. Example:

```
      int x = 1;
      int a = 2;
      int b = 3;
      x?a:b = 1;
-------------^------------------ERROR
```

- Assignment operators are incorrectly implemented. Example:

```
      (i = 2) = 3;
-------------^-------- The result of the = operator shall be an lvalue
      (i *= 2) = 3;
-------------^-------- The result of the *= operator shall be an lvalue
```

```
      (i += 5) = 3;
-------------^-------- The result of the += operator shall be an lvalue
```

24.2.3 Equality Operators

The following equality operator features are unsupported.

- Defining a pointer to member function type. Example:

```
struct X {
      void f() {}
};
typedef void (X::*PROC)();
```

- Permitting an implementation to compare a pointer to member operand with a constant expression which evaluates to zero using the == operator.

```
class X {
public:
      int m;
};
(void) ( &X::m == 0 );
 -----------^-------------ERROR
```

24.3 Header Files

Header files of type `std namespace` are unsupported.

Included `cname` header files are not mapped to `name.h`. Example:

```
#include <cstring>
-------^-------------------- ERROR
```

The following table shows unimplemented header files.

Table 24-1. Unimplemented Header Files

<algorithm>	<iomanip>	<memory>	<streambuf>
<bitset>	<iosfwd>	<new>	<typeinfo>
<climits>	<iostream>	<numeric>	<utility>
<complex>	<istream>	<ostream>	<valarray>
<deque>	<iterator>	<queue>	<vector>
<exception>	<limits>	<sstream>	<wchar.h>
<fstream>	<list>	<stack>	<wctype.h>
<functional>	<map>	<stdexcept>	

24.4 Bigraph and Trigraph Support

The compiler does not recognize the trigraph sequence `??!` as equal to `|`.

In some cases the compiler fails to replace the `%:` sequence. Example:

```
#if (4 == 9)
#include <string.h>
%:endif
^---------------------------- ERROR (missing endif directive)
```

24.5 Known Class Issues

The following section describes known class issues and unimplemented or unsupported features.

- Class Names

  Usually, using elaborate type specifiers ensures the validity of both names when you define a class and a function with the same name in the same scope. However, in the HC(S)08 compilers this type of class name definition causes an error. Example:

```
class C { char c; };

void C(int x) { }
```

```
int x;

void main()

{

      C(x);

------^----------------------- ERROR

}
```

- Local classes are unsupported on the HC(S)08 compilers. Example:

```
void f(void)

{

  class C {

    C() { }

  };

}
```

- The class member access feature is unsupported. Example:

```
      class X {

      public:

             enum E { a, b, c };

      } x;

      int type(int ) {return INT;}

      int type(long ) {return LONG;}

      int type(char ) {return CHAR;}

      int type(X::E ) {return ENUMX;}

      type(x.a);

----------^---------------- Ambiguous parameters type
```

- Nested class declaration is unsupported, although some accesses and calls may succeed when using nested classes.
- Nested class depths of ten or more are not supported. Example:

```
struct :: A a;

-------------------^----------------ERROR
```

- Function member definitions are not allowed within local class definitions. Example:

```
void f (){

  class A{

        int g();

---------------^------Illegal local function definition

  };

}
```

- Defining a class within a function template is not allowed. Example:

```
template <class T>

struct A {

        void f();

};

template <class T>

void A<T>::f(){

        class B {

                T x;

        };

-------------^----------------ERROR

}
```

- Unsupported Scope rules for classes

Declaring the name of a class does not ensure that the scope name extends through the declarative regions of classes nested within the first class. Example:

```
struct X4 {

      enum {i = 4};

      struct Y4 {

            int ar[i];

---------------^-------------------ERROR

      }

}
```

- Unimplemented Storage class specifiers

  Normally, C++ allows taking the address of an object declared register. Example:

```
register int a;

int* ab = &a;

----------^----- ERROR: Cannot take address of this object
```

- The `mutable` storage class specifier is unsupported.

24.6 Keyword Support

The following keywords are unsupported:

- `typeid`
- `explicit`
- `typename`
- `mutable` storage class specifier
- Cast keywords:
  - `static_cast`
  - `const_cast`
  - `reinterpret_cast`
  - `dynamic_cast`

24.7 Member Issues

The following member features are either unimplemented, unsupported, or not functioning correctly in the HC(S)08 compilers.

- Pointer to Member
  - Global pointer to member initialization is unimplemented. Example:

```
struct S1{};
struct S2 { int member; };
struct S3 : S1, S2 {};
int S3::*pmi = &S3::member;
-------------^-------------- ERROR
```

  - Accessing or initializing a class member using a `pointer_to_member` from that class is unsupported. Example:

```
class X{
public :
   int a;
};
int main(){
   int X::* p0 = &X::a;
   X obj;
   obj.*p0 = -1;
--------^-----------------ERROR:Unrecognized member
}
```

  - Constructing an array from a pointer to member of a struct is unsupported. Example:

```
int S::* a0[3];
a0[1] = &S::i
---------^-----------------Failed
```

- Static member - When you refer a static member using the class member access syntax, the object-expression is not evaluated or is evaluated incorrectly. Example:

```
int flag;
struct S {
      static int val(void) { return flag; }
} s;
S* f01() { flag = 101; return &s; }
void main(){
      int g;
      g = f01()->val(); //evaluation failed
}
```

- Non-Static Member Functions
  - Using non-static data members defined directly in their overlying class in non-static member functions is unsupported. Example:

```
class X {
      int var;
public:
      X() : var(1) {}
      int mem_func();
} x;
```

```
int X::mem_func(){
      return var; //returned value should be 1
}
```

- A non-static data member/member function name should refer to the object for which it was called. However, in the HC(S)08 compiler, it does not. Example:

```
class X {
public:
     int  m;
     X(int a) : m(a) {}
}
X obj = 2;
 int a  = obj.m; //should be 2 (but is not)
```

- Member Access Control
  - Accessing a protected member of a base class using a friend function of the derived class is unsupported. Example:

```
class A{
protected:
      int i;
};
class B:public A{
      friend int f(B* p){return p->i};
} ;
```

  - Specifying a private nested type as the return type of a member function of the same class or a derived class is unsupported. Example:

```
class A {
protected:
      typedef int nested_type;
      nested_type func_A(void);
};
Class B: public A{
      nested_type func_B(void);
};
A::nested_type A::func_A(void) { return m; }
B:: nested_type B::func_B(void) { return m; }
^-----------------------------------ERROR: Not allowed
```

  - Accessing a protected member is unsupported. Example:

```
class B {
protected:
      int i;
};
class C : private B {
      friend void f(void);
};
void f(void) { (void) &C::i;}
-------------------------^------ERROR: Member cannot be accessed
```

  - Access declaration

    Base class member access modification is unimplemented in the following case:

```
class A{
public:
      int z;
};
class B: public A{
public:
      A::z;
---------^------------ERROR
};
```

24.8 Constructor and Destructor Functions

The compiler does not support the following destructor features:

- When a class has a base class with a virtual destructor, its user-declared destructor is virtual
- When a class has a base class with a virtual destructor, its implicitly-declared destructor is virtual

The compiler does not support the following constructor features:

- Copy constructor is an unsupported feature. Example:

```
class C { int member;};

void f(void) {

    C c1;

    C c2 = c1;

-------^-----------ERROR: Illegal initialization of non-aggregate type

}
```

- Using a non-explicit constructor for an implicit conversion (conversion by constructor) is unsupported. Example:

```
class A{

public:

      int m;

      S(int x):m(x){};

};

int f(A a) {return a.m};

int b = f(5) /*value of b should be 5 because of explicit conversion of f parameter(b =
f(A(5)))*/
```

- Directly invoking a virtual member function defined in a derived class using a constructor/destructor of class x is unsupported. Example:

```
class A{

      int m;

      virtual void vf(){};

      A(int) {vf()}

}

class B: public A{

      void vf(){}

      B(int i) : A(i) {}

}

B b(1); // this should result in call to A::vf()
```

- Indirectly invoking a virtual member function defined in a derived class using a constructor of class x is unsupported. Example:

```
class A{

      int m;

      virtual void vf(){};

      void gf(){vf();}

      A(int) {gf();}

}

class B: public A{

      void vf(){}

      B(int i) : A(i) {}

}
```

```
B b(1); // this should result in call to A::vf()
```

- Invoking a virtual member function defined in a derived class using a `ctor-initializer` of a constructor of class x is unsupported. Example:

```
class A{

      int m;

      virtual int vf(){return 1;};

      A(int):m(vf()){}

}

class B: public A{

      int vf(){return 2;}

      B(int i) : A(i) {}

}

B b(1); // this should result in call to A::vf()
```

24.9 Overload Features

The following overload features are unsupported at this time.

- Overloadable Declarations

  Usually, two function declarations of the same name with parameter types that only differ in a parameter that is an enumeration in one declaration, and a different enumeration in the other, can be overloaded. This feature is unsupported at this time. Example:

```
      enum e1 {a, b, c};
      enum e2 {d, e};
      int g(e1) { return 3; }
      int g(e2) { return 4; }
----------------^--------------------ERROR:function redefinition
```

- Address of Overloaded Function

  Usually, in the context of a pointer-to-function parameter of a user-defined operator, using a function name without arguments selects the non-member function that matches the target. This feature is unsupported at this time. Example:

```
const int F_char  = 100;
int func(char)
{
      return F_char;
}
struct A {} a;
int operator+(A, int (*pfc)(char))
{
      return pfc(0);
}
if (a + func != F_char){}
-----------^----------------- Arithmetic types expected
```

- Usually, in the context of a pointer-to-member-function return value of a function, using a function name without arguments selects the member function that matches the target. This feature is unsupported at this time. Example:

```
struct X {
      void f (void)  {}
      void f (int) {}
} x;
typedef void (X::*mfvp)(void);
mfvp f03() {
              return &X::f;
----------------------^-------ERROR:Cannot take address of this object
}
```

- Usually, when an overloaded name is a function template and template argument deduction succeeds, the resulting template argument list is used to generate an overload resolution candidate that should be a function template specialization. This feature is unsupported at this time. Example:

```
template <class T> int f(T) { return F_char; }
int f(int) { return F_int; }
```

```
int (*p00)(char) = f;

--------------------------------^-----------ERROR: Indirection to different types ('int
(*)(int)' instead of 'int (*)(char )')
```

- Overloading operators is unsupported at this time. Example:

```
struct S {

      int m;

      template <class T> void operator+=(T t) { m += t; } //

ERROR at template

};
```

24.10 Conversion Features

The following conversion features are unsupported.

- Implicit conversions using non-explicit constructors are unsupported. Example:

```
class A{

public:

      int m;

      S(int x):m(x){};

};

int f(A a) {return a.m};

int b = f(5) /*value of b should be 5 because of explicit conversion of f parameter(b =
f(A(5)))*/
```

- Initializations using user-defined conversions are unsupported. Usually, when you invoke a user-defined conversion to convert an assignment-expression of type `cv S` (where `S` is a class type), to a type `cv1 T` (where `T` is a class type), a conversion member function of `S` that converts to `cv1 T` is considered a candidate function by overload resolution. However, this type of situation is unsupported on HC(S)08 compilers. Example:

```
struct T{

      int m;

      T() { m = 0; }

} t;

struct S {

      operator T() { counter++; return t; }
```

```
} s00;

T t00 = s00;
---------------^------Constructor call with wrong number of arguments
```

The following topics are covered here:

- Standard Conversion Sequences
- Ranking Implicit Conversion Sequences
- Explicit Type Conversion

24.10.1 Standard Conversion Sequences

The following standard conversion sequences are unsupported:

- A standard conversion sequence that includes a conversion having a conversion rank. Example:

```
int f0(long double) { return 0; }
int f0(double) { return 1; }
float f = 2.3f;
value = f0(f); //should be 1
-----------^------------- ERROR ambiguous
```

- A standard conversion sequence that includes a promotion, but no conversion, having a conversion rank. Example:

```
int f0(char) { return 0; }
int f0(int) { return 1; }
 short s = 5;
value = f0(s);
-----------------^------------- ERROR ambiguous
```

- A pointer conversion with a Conversion rank. Example:

```
int f0(void *) { return 0; }
int f0(int) { return 1; }
value = f0((short) 0);
----------------^------------- ERROR ambiguous
```

- User-Defined Conversion Sequences

A conversion sequence that consists of a standard conversion sequence, followed by a conversion constructor and a standard conversion sequence, is considered a user-defined conversion sequence by overload resolution and is unsupported. Example:

```
char k = 'a';
char * kp = &k;
struct S0 {
      S0(...) { flag = 0; }
      S0(void *) { flag = 1; }
};
const S0& s0r = kp;
-----------------^-----ERROR: Illegal cast-operation
```

24.10.2 Ranking Implicit Conversion Sequences

The following implicit conversion sequence rankings situations are unsupported at this time.

- When `s1` and `s2` are distinct standard conversion sequences and `s1` is a sub-sequence of `s2`, overload resolution prefers `s1` to `s2`. Example:

```
int f0(const char*) { return 0; }
int f0(char*) { return 1; }
value = f0('a');
---------------^----------------ERROR:Ambiguous
```

- When `s1` and `s2` are distinct standard conversion sequences of the same rank, neither of which is a sub-sequence of the other, and when `s1` converts `c*` to `b*` (where b is a base of class c), while `s2` converts `c*` to `a*` (where `a` is a base of class `b`), then overload resolution prefers `s1` to `s2`. Example:

```
struct a
struct b : public a
struct c : public b
int f0(a*) { return 0; }
int f0(b*) { return 1; }
c* cp;
value = f0(cp);
```

```
-------------^---------------ERROR:Ambiguous
```

- When s1 and s2 are distinct standard conversion sequences neither of which is a subsequence of the other, and when s1 has Promotion rank, and s2 has Conversion rank, then overload resolution prefers s1 to s2. Example:

```
int f(int) { return 11; }

int f(long) { return 55; }

short aa = 1;

int i = f(aa)

----------^------------------- ERROR:Ambiguous
```

24.10.3 Explicit Type Conversion

The following syntax use is not allowed when using explicit type conversions on an HC(S)08 compiler:

```
i = int();//A simple-type-name followed by a pair of parentheses
```

The following explicit type conversion features are unsupported at this time:

- Casting reference to a volatile type object into a reference to a non-volatile type object. Example:

```
volatile int x = 1;

volatile int& y= x;

if((int&)y != 1);

-------^-------------------------ERROR
```

- Converting an object or a value to a class object even when an appropriate constructor or conversion operator has been declared. Example:

```
class X {

public:

      int i;

      X(int a) { i = a; }
```

```
};

   X x = 1;

   x = 2;

-----^------------------ERROR: Illegal cast-operation
```

- Explicitly converting a pointer to an object of a derived class (private) to a pointer to its base class. Example:

```
class A {public: int x;};

class B : private A {

public:

      int y;

};

int main(){

      B b;

      A *ap = (A *) &b;

---------------^----- ERROR: BASE_CLASS of class B cannot be accessed

}
```

24.11 Initialization Features

The compiler does not support the following initialization features:

- When an array of a class type `T` is a sub-object of a class object, each array element is initialized by the constructor for `T`. Example:

```
class A{

public:

      A(){}
```

```
};

class B{

public:

      A x[3];

      B(){};

};

B b; /*the constructor of A is not called in order to initialize the elements of the
array*/
```

- Creating and initializing a new object (call constructor) using a new-expression with one of the following forms:
  - `(void) new C();`
  - `(void) new C;`
- When initializing bases and members, a constructor's `mem-initializer-list` may initialize a base class using any name that denotes that base class type (`typedef`); the name used may differ from the class definition. Example:

```
struct B {

      int im;

      B(int i=0) { im = i; }

};

typedef class B B2;

struct C : public B {

      C(int i) : B2(i) {} ;

---------------------^------------------ERROR

};
```

- Specifying explicit initializers for arrays is not supported. Example:

```
typedef M MA[3];

struct S {

      MA a;
```

```
      S(int i) : a() {}

----------------------^----------ERROR: Cannot specify explicit initializer for arrays

};
```

- Initialization of local static class objects with constructor is unimplemented. Example:

```
struct S {

      int a;

      S(int aa) : a(aa) {}

};

static S s(10);

---------^-------------ERROR
```

See Conversion Features also.

24.12 Known Errors

The following functions are incorrectly implemented:

- `sprintf`
- `vprintf`
- `putc`
- `atexit from stdlib.h`
- `strlen from string.h`
- IO functions (`freopen`, `fseek`, `rewind`, etc.)

The following errors occur when using C++ with the HC(S)08 compiler.

- EILSEQ is undefined when `<errno.h>` is included
- Float parameters pass incorrectly

```
int func(float, float, float );

func(f, 6.000300000e0, 5.999700000e0)
```

```
the second value becomes -6.0003
```

- Local scope of `switch` statement is unsupported for the default branch. Example:

```
switch (a){
      case 'a': break;
      default :
            int x = 1;
      ----------^--------------ERROR: Not declared x
}
```

- An `if` condition with initialized declaration is unsupported. Example:

```
if(int i = 0)
------^-----------------ERROR
```

The following internal errors occur when using C++ with the HC(S)08 compiler:

- Internal Error #103. Example:

```
long double & f(int i ) {return 1;}
long double i;
if (f(i)!=i)
--------^-----------------Internal Error
```

- Internal Error #385, generated by the following example:

```
class C{
public:
      int n;
      operator int() { return n; };
}cy;
switch(cy) {
```

```
--------^-------------ERROR

    case 1:

      break;

    default:

        break;

}
```

- Internal Error #418, generated by the following example:

```
#include <time.h>

struct std::tm T;
```

- Internal Error #604, generated by the following example:

```
class C {

      public:

              int a;

            unsigned func() { return 1;}

};

unsigned (C::*pf)() = &C::func;

if (pf != 0 );

-------^------------------Generates the error
```

- Internal Error #1209, when using a twelve-dimensional array
- Internal Error #1810, generated by the following example:

```
struct Index {

      int s;

      Index(int size) { s = size; }

      ~Index(void){ ++x; }

};
```

```
for (int i = 0; i < 10; i++)

      for (Index j(0); j.s < 10; j.s++) {

            // ...

      }
```

24.13 Other Features

This section describes unsupported or unimplemented features.

- Unsupported data types include:
  - `bool`
  - `wchar_t` (wide character).
- Exception h andling is unsupported
- Using comma e xpressions as `lvalues` is unsupported. Example:

```
(a=7, b) = 10;
```

- Name Features
  - Namespaces are currently unsupported. Example:

```
namespace A {
----------^------------------- ERROR
  int f(int x);
}
```

  - The name lookup feature is currently unsupported. Name lookup is defined as looking up a class as if the name is used in a member function of X when the name is used in the definition of a static data member of the class. Example:

```
class C {
public:
      static int i;
      static struct S {
      int i; char c;
      } s;
};
int C::i = s.i;
```

  - Hiding a class name or enumeration name using the name of an object, function, or enumerator declared in the same scope is unsupported. Example:

```
enum {one=1, two, hidden_name };
struct hidden_name{int x;};
-----------^----------------Not allowed
```

- Global initializers with non-`const` variables are unsupported. Example:

```
int x;
int y = x;
```

- Anonymous unions are unsupported. Example:

```
void f()
{
      union { int x; double y; };
      x = 1;
      y = 1.0;
}
```

- The following time functions (<ctime>) are unsupported:
  - time()
  - localtime()
  - strftime()
  - ctime()
  - gmtime()
  - mktime()
  - clock()
  - asctime()
- The fundamental type feature is not supported:

```
int myfun (char x){}
int myfun (unsigned char x){}
-------------^-------------------------Illegal function redefinition
```

- Enumeration declaration features
  - Defining an enum in a local scope of the same name is unsupported. Example:

```
enum e { gwiz };  // global enum e
void f()
{
enum e { lwiz };
---------------^---------------- ERROR: Illegal enum redeclaration
}
```

  - The identifiers in an enumerator-list declared as constants, and appearing wherever constants are required, is unsupported. Example:

```
int myfun(short l) { return 0; }
int myfun(const int l) { return 1; }
enum E { x, y };
myfun(x); /*should be 1*/
```

- Unsupported union features:
  - An unnamed union for which an object is declared having member functions
  - Allocation of bit-fields within a class object. Example:

```
enum {two = 2};
struct D { unsigned char : two; };
```

- The following multiple base definition features are unimplemented as yet:
  - More than one indirect base class for a derived class. Example:

```
Class B:public A(){};
Class C: public B(){};
Class D :public B, public A,publicC{};
```

  - Multiple virtual base classes. Example:

```
class A{};
class B: public virtual A{};
class C: public virtual A{};
class D: public B, public C{}
```

- Generally, a friend function defined in a class is in the scope of the class in which it is defined. However, this feature is unsupported at this time. Example:

```
class A{
public:
      static int b;
      int f(){return b;};
};
int A::b = 1;
int x = f(); /*ERROR : x!=1 (it should be 1)*/
```

- The compiler considers the following types ambiguous (the same):
  - `char`
  - `unsigned char`
  - `signed char`
- The Call to Named Function feature is unsupported. Example:

```
class A{
      static int f(){return 0;}
     friend void call_f(){
            f();
      ------^-----ERROR: missing prototype (it should be accepted                    by
the compiler)
      }
}
```

- Preprocessing directives are unsupported. Example:

```
#define  MACRO  (X) 1+ X
MACRO(1) + 1;
-------------^--------------------Illegal cast-operation
```

- The following line control feature is unsupported.
  - Including a character-sequence in a line directive makes the implementation behave as if the content of the character string literal is equal to the name of the source file. Example:

```
#line 19 "testfile.C" //line directive should alter __FILE__
```

- The following floating point characteristics errors occur:
  - Float exponent is inconsistent with minimum

```
power(FLT_RADIX, FLT_MIN_EXP -1) != FLT_MIN
```

  - Float largest radix power is incorrect

```
FLT_MAX / FLT_RADIX + power(FLT_RADIX, FLT_MAX_EXP-FLT_MANT_DIG-1)!=
power(FLT_RADIX,FLT_MAX_EXP-1)
```

  - Multiplying then dividing by radix is inexact
  - Dividing then multiplying by radix is inexact
  - Double exponent is inconsistent with minimum
  - Double, power of radix is too small
  - Double largest radix power is incorrect
  - Multiplying then dividing by radix is inexact
  - Dividing then multiplying by radix is inexact

    - Long double exponent is inconsistent with minimum
    - Long double, power of radix is too small
    - Long double largest radix power is incorrect
- The following best viable function is unsupported:
    - When two viable functions are indistinguishable implicit conversion sequences, it is normal for the overload resolution to prefer a non-template function over a template function. Example:

```
int f ( short , int ) { return 1; }
template <class T> int f(char, T) { return 2; }
value = f(1, 2);
---------^----------------------ERROR: Ambiguous
```

- The following Reference features are unsupported:
    - Object created and initialized/destroyed when reference is to a `const`. Example:

```
const X& r = 4;
-----------------------^------------ERROR: Illegal cast-operation
```

    - The following syntax is unsupported:

```
int a7, a;
if(&(::a7) == &a);
---------^-------------------ERROR:Not supported operator ::
```

- Aggregate features
    - Object initialization fails. Example:

```
class complex{
      float re, im;
      complex(float r, float i = 0) { re=r; im=i; };
      int operator!=( complex x ){}
}
complex z = 1;
z!=1
---------^------------ERROR :Type mismatch
```

    - Initialization of aggregate with an object of a struct/class publicly derived from the aggregate fails. Example:

```
class A {
      public:
int a;
A(int);
 };
class B: public A{
 public:
       int b;
      B(int, int);>
};
B::B(int c, int d) : A(d) { b = c; }
 B b_obj(1, 2);
int x = B_obj.a;
-----^----------ERROR: x should be 2
```

- Evaluating default arguments at each point of call is an unsupported feature.
- The following typedef specifier is unsupported:

```
typedef int  new_type;
typedef int  new_type;
--------------^-------ERROR: Invalid redeclaration of new_type
```

- This return statement causes an error:

```
return ((void) 1);
--------------------------^-----------ERROR
```

- Permitting a function to appear in an integral constant if it appears in a `sizeof` expression is unsupported. Example:

```
void f() {}
int i[sizeof &f];
-------------------^-------------ERROR
```

- Defining a local scope using a compound statement is an unimplemented feature. Example:

```
int i = 4;
int main(){
      if ((i != 1) || (::i != 4));
------------------------^----------ERROR
}
```

- The following Main function is currently unimplemented:

```
argv[argc]!=0 (it should be guaranteed that argv[argc]==0.)
```

- The following Object lifetime feature is currently unimplemented:
  - When the lifetime of an object ends and a new object is created at the same location before it is released, a pointer that pointed to the original object can be used to manipulate the new object.
- The following function call features are unsupported:
  - References to functions feature is not supported. Example:

```
int main(){
      int f(void);
int (&fr)(void) = f;/
}
```

  - Return pointer type of a function make ambiguous between `void *` and `X *`. Example:

```
class X {
public:
      X *f() { return this; }
};
int type(void *x) {return VOIDP;}
int type(X *x) {return CXP;}
X x;
type(x.f())
-----^--------ERROR: ambiguous
```

  - Incorrect implementation of a member function call when the call is a conditional expression followed by argument list. Example:

```
struct S {
S(){}
      int f() { return 0; }
      int g() { return 11; }
int h() {
            return (this->*((0?(&S::f) : (&S::g))))();
------------------------------^-------------ERROR
      }
};
```

- The following Enumeration feature is unsupported:
  - For enumerators and objects of enumeration type, if an `int` can represent all the values of the underlying type, the value is converted to an `int`; otherwise if an

`unsigned int` can represent all the values, the value is converted to an `unsigned int`; otherwise if a `long` can represent all the values, the value is converted to a `long`; otherwise it is converted to `unsigned long`. Example:

```
enum E { i=INT_MAX, ui=UINT_MAX , l=LONG_MAX, ul=ULONG_MAX };
------------------------^-------------ERROR: Integral type expected or enum value
out of range
```

- Delete operations have the following restrictions:
  - Use the `S::operator` delete only for single cell deletion and not array deletion. For array deletion, use the global `::delete()`. Example:

```
struct S{
      S() {}
      ~S () {destruct_counter++;}
      void * operator new (size_t size) {
            return new char[size];
      }
      void operator delete (void * p) {
            delete_counter ++;
            ::delete p;}
      };
      S * ps = new S[3];
      delete [] ps;
-------------^--------ERROR: Used delete operator (should use global ::delete)
```

  - Global `::delete` uses the class destructor once for each cell of an array of class objects. Example:

```
  S * ps1 = new S[5];
  ::delete [] ps1;
------------^------ERROR: ~S is not used
```

  - Error at declaring delete operator. Example:

```
void operator delete[](void *p){};
----------------------^-------------ERROR
```

- The New operator is unimplemented. Example:

```
- void * operator new[](size_t);
-------------------^--------ERROR: Operator must be a function
```

- The following Expression fails to initialize the object. Example:

```
int *p = new int(1+(2*4)-3);
-----------------^-----ERROR: The object is not initialized
```

- Use placement syntax for new `int` objects. Example:

```
int * p1, *p2;
p1 = new int;
p2 = new (p1) int;
--------------^----------------ERROR: Too many arguments
```

- The following Multi-dimensional array syntax is not supported:

```
int  tab[2][3];
int myfun(int (*tab)[3]);
-------------------^---------------------ERROR
```

- The following `goto` or `switch` syntax is unsupported:

```
label:
int x = 0;
-------------^--------------ERROR: x not declared (or typename)
```

- The following Declaration Statement feature is not implemented:
  - Transfer out of a loop, out of a block, or past an initialized `auto` variable involves the destruction of `auto` variables declared at the point transferred from but not at the point transferred to.
- The following Function Syntax features are not supported:
  - Function taking an argument and returning a pointer to a function that takes an integer argument and returns an integer should be accepted. Example:

```
int (*fun1(int))(int a) {}
int fun2(int (*fun1(int))(int))()
-----^-------------------------ERROR
```

  - Declaring a function `fun` taking a parameter of type integer and returning an integer with typedef is not allowed. Example:

```
typedef int fun(int)
------------------------^-----ERROR
```

  - A `cv-qualifier-seq` can only be part of a declaration or definition of a non-static member function, and of a pointer to a member function. Example:

```
class C {
      const int fun1(short);
      volatile int fun2(long);
      const volatile int fun3(signed);
};
const int (C::*cp1)(short);
-------------^--------------- ERROR:Should be initialized
volatile int (C::*cp2)(long);
-------------^--------------- ERROR: Should be initialized
const volatile int (C::*cp3)(signed);
------------------------^---- ERROR: Should be initialized
```

  - Use of `const` in a definition of a pointer to a member function of a struct should be accepted. Example:

```
struct S {
     const int fun1(void);
      volatile int fun2(void);
      const volatile int fun3(void);
} s;
const int (S::*sp1)(void) = &S::fun1;
      if(!sp1);
--------^---------------------------ERROR:Expected int
```

- When using Character literals, the Multi-characters constant is not treated as `int`. Example:

```
int f(int i, char c) {return 1;}
f('abcd', 'c');
-----------^--------------------ERROR
```

- The String characteristic "A string is an `array of `n const char`'" is not supported. Example:

```
int type(const char a[]){return 1};
type("five") != 1 /*Runtime failed*/
```

- Ambiguity Resolution

```
struct S {
      int i;
      S(int b){ i = b;}
```

```
};
S x(int(a));
--------^----------ERROR: Should have been a function declaration, not an object
declaration
```

- Using `const` as a qualified reference is an unsupported feature. Example:

```
int i;
typedef int& c;
const c cref = i;// reference to int
--------------------^-----------------ERROR
```

- No warning on invalid jump past a declaration with explicit or implicit initializer. Example:

```
switch(val)
{
case 0:
int a = 10; // invalid, warning should be reported
break;
case 1:
int b; // valid
b = 11;
break;
case 2:
break;
}
```

Chapter 25
Banked Memory Support

This appendix describes the architecture of the Banked memory support and introduces you to the Paged memory and Linear address space in the HCS08 family.

The topics covered here are as follows:

- Introduction
- Paged Memory and Non-Paged Memory
- Linear Memory Space

25.1 Introduction

Having an address bus of 16-bits wide, the typical HCS08 Core architecture limits the available CPU addressable space to 64K bytes. However, some special HCS08 MCUs, like the MC9S08QE128 which contains a MMU (Memory Management Unit), extends the processor's ability of addressing up to 4M memory space.

On these derivatives, memory can be accesses using the paging mechanism, using the linear memory access mechanism or directly, if the address is below 64K.

To avoid common pitfalls and to detect where there may be room for code optimizations, you need to have a complete understanding of your application. Thus, it is essential to know how to correctly access a given memory location and how CodeWarrior distributes your code between these three memory categories.

The following sections in this chapter introduce the notions of these three kinds of memory and highlight their differences.

25.2 Paged Memory and Non-Paged Memory

As mentioned above, the typical HCS08 Core architecture can only access 2^16 = 65536 bytes, or 64 Kilobytes memory space. Thus, a paging mechanism was introduced to enable HCS08 CPU access more memory space. The terms "paged" and "non-paged" are derived from the paging mechanism. "Banked" and "non-banked" are synonym terms to paged and non-paged, respectively. They are often used interchangeably in Freescale's literature.

To better understand paging mechanism, we need to consider the following:

- Notion of Local Map
- Notion of Page Window
- Notion of Memory Page
- Page-switching Mechanism
- Compiler Support
- Example

25.2.1 Notion of Local Map

The term "CPU local map" refers to the 64K space that the CPU can directly - or naturally access through its instruction set with a 16-bit address.

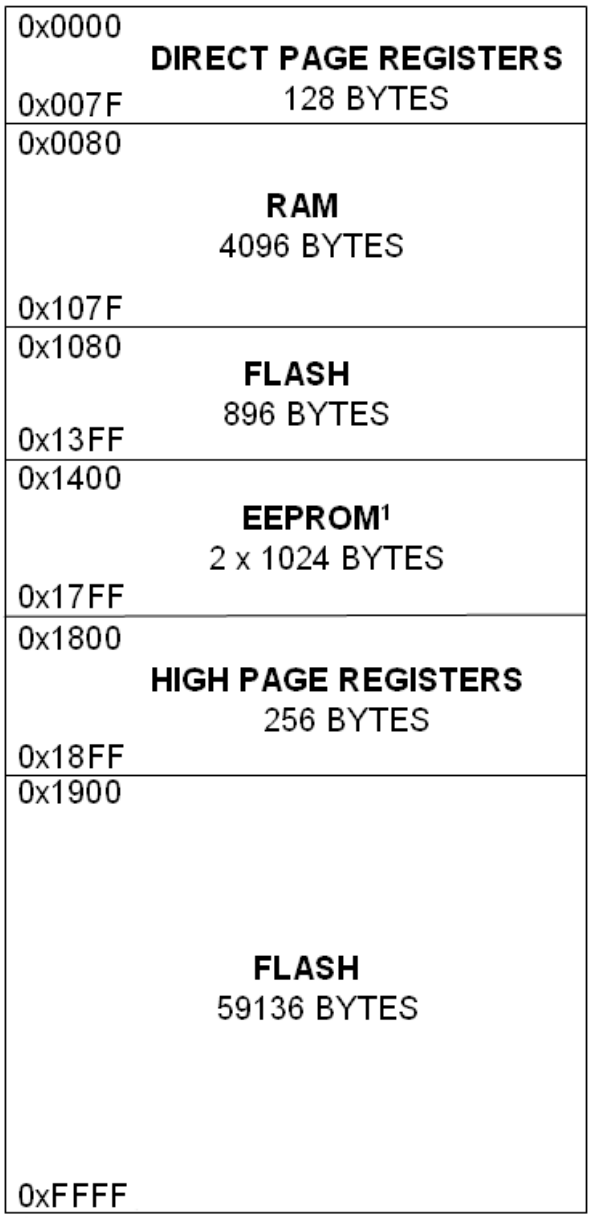

Figure 25-1. CPU Local Map for an S08 Device

We can see in the above figure that the registers and the other memory resources have dedicated address ranges. The above figure illustrates the local map for an MC9S08DZ60 part.

In the case of the HCS08 family, local memory maps may change from device to device, however, they do share a common characteristic. The common characteristic is that RAM, EEPROM and Register space boundaries may change from device to device but are always by default located in the first 16K region of the local map (from addresses 0x0000 to 0x3FFF) . The upper 48K region is devoted to hosting Flash memory (From address 0x4000 to 0xFFFF).

When paging mechanism is applied to extend the memory space, the 48K region is divided into three 16K regions. The middle region 16K region (from 0x8000 to 0xBFFF) is called the Flash page window, plays a very special role in paging mechanism.

25.2.2 Notion of Page Window

Most of the local addresses in the CPU local map always "point" to well-defined fixed physical locations. Certain local addresses, however, will not always point to the same physical locations. These special address ranges are called "page windows". Local addresses inside a page-window range are addresses whose mere 16-bits are not enough information for the MMU to determine a well-defined physical location.

For a local address inside a page window range, the MMU requires additional information, stored inside a register, to be able to translate the given local address into a well-defined physical location. Such a register is called a Page-register.

25.2.3 Notion of Memory Page

A memory "page" is simply a continuous section of physical memory that has a fixed size. In the case of HCS08 family, only the flash resource is paged. For instance, for MC9S08QE128 the page size is 16K. The size of a flash page is 16K. Note that the size of a memory page is the same as the size of that memory resource's page window in the local map.

The division of physical memory into pages is just a conceptual division. Pages do not correspond to real physical divisions of memory. Once conceptual division is done, a number is assigned to each page.

25.2.4 Page-switching Mechanism

In order to view a particular physical page in the local map's page window, the programmer needs to write the page number into the page register.

Once the appropriate page of physical memory has been "displayed" inside the page window, the CPU can access this data with a 16 bit address corresponding to a location inside the page window.

It is important to understand that, when a memory resource is paged, the totality of this resource can be accessed through the page window. However, writing to the page register every time a memory location has to be accessed introduces a certain amount of overhead. This is why, certain locations are also mapped directly into the local map.

These locations that are always mapped onto the CPU local map regardless of any page-register value are called non-banked, or non-paged locations. For these locations, most of the time the paged access is not used and the direct access is always preferred. The totality of a paged-memory resource is conceptually divided into numbered pages, non banked locations from the resource and have a certain page number associated.

The fiugre below shows the page numbers that are associated with the non-banked locations in the local map and page numbers that associated with the extended locations. For local map locations, it can be accessed by two ways.

For example, address 0x038000 is equivalent to address 0xC000.

The following figure shows the paging mechanism of device MC9S08QE128.

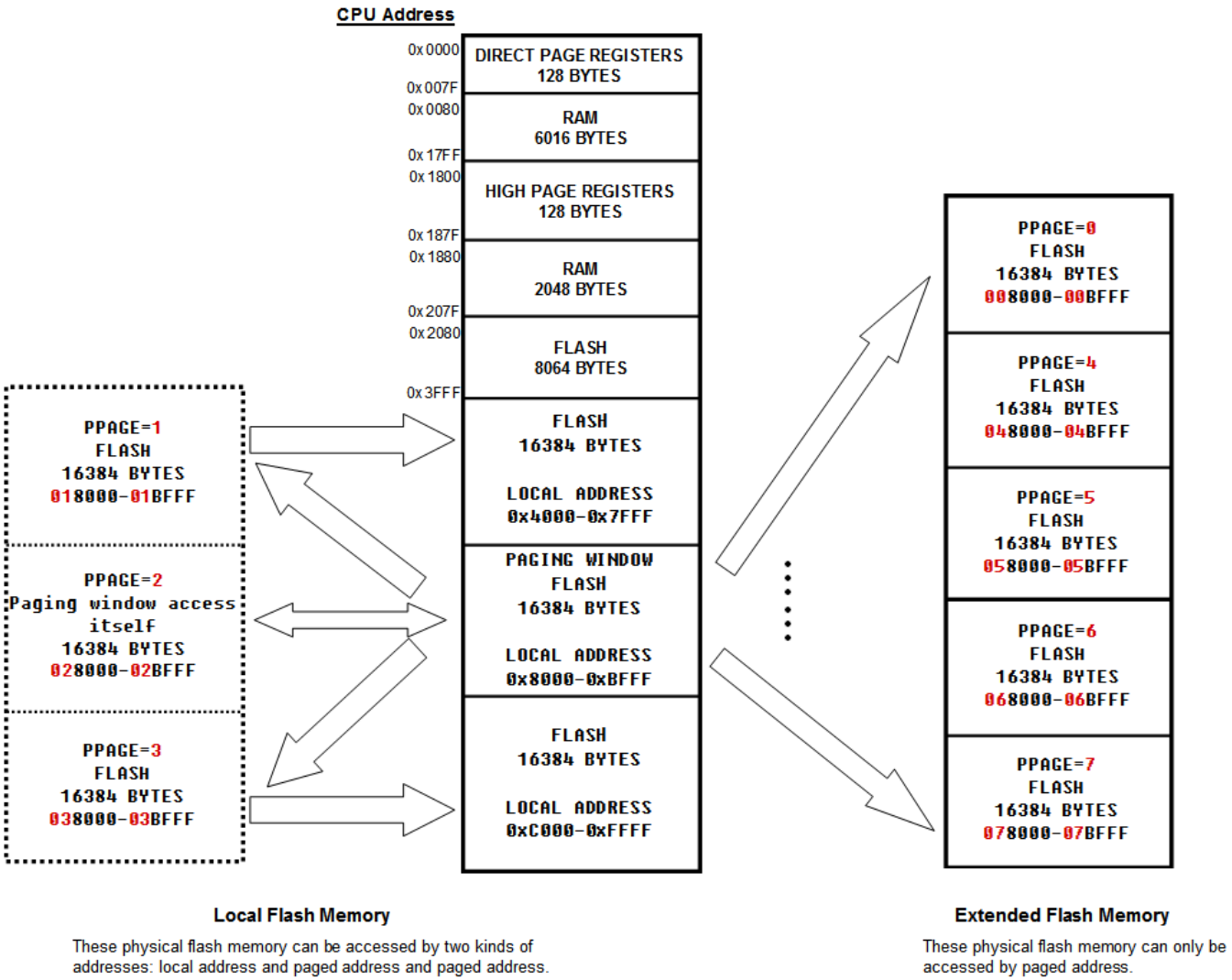

Figure 25-2. MC9S08QE128 Memory Map

25.2.5 Compiler Support

CodeWarrior's C compiler takes care of automatically inserting instructions that write the appropriate value to the page register.

How this is accomplished differs for code and data. Functions placed in the extended memory are usually defined using the `__far` keyword (in the banked memory model all the functions are by default `__far`); data placed in the extended memory can be accessed using linear addressing (via `__linear` pointers for instance).

To ensure that this happens, the programmer needs to select the memory model that is most appropriate for the application, and use special qualifiers like `__near` or `__far` keywords, or `#pragma` statements to locally modify compiler behavior where needed.

The `__far` keyword when associated to a function (for example. `void __far f()`) instructs the compiler to consider that function as placed in the extended memory.

The `__near` keyword when associated to a function (for example. `void__near f()`) instructs the compiler to consider that function as placed inthe non-banded memory.

Near functions can also be placed in the banked memory, but only when they are called by other functions placed in the same page.

Table 25-1. Memory Models and __near and __far Keywords

Tiny Memory Model	Small Memory Model	Banked Memory Model
Functions are inherently __near if not specified otherwise	Functions are inherently __near if not specified otherwise	Functions are inherently __far if not specified otherwise
Functions in extended memory need to be marked with __far (or #pragma CODE_SEG __FAR_SEG)	Functions in extended memory need to be marked with __far (or #pragma CODE_SEG __FAR_SEG)	Functions that do not reside in the extended memory (or those that can use the "classical" calling convention) need to be marked with __near (or #pragma CODE_SEG __NEAR_SEG)

25.2.6 Example

A simple example illustrates placing code in the extended memory.

Listing: Placing Code in the Extended Memory

```
Place function f() in a far segment P5.
#pragma CODE_SEG __FAR_SEG P5
void f()
{
  ...
}
```

In PRM file, place segment P5 at the correct address.

```
//PRM file
SEGMENTS
...
PPAGE_5     =  READ_ONLY     0x058000 TO 0x05BFFF;
...
END
PLACEMENT
...
```

```
P2      INTO    PPAGE_5;
...
END
```

25.3 Linear Memory Space

Some HCS08 derivatives (with an MMU) provide a special mechanism to access data. This mechanism makes use of addresses that run in a linear fashion, that is, memory is treated as one contiguous memory space.

This section covers the following topics:

- Notion of Linear Memory Space
- Compiler Support
- Example

25.3.1 Notion of Linear Memory Space

Refer the following figure for paged/linear address equivalence.

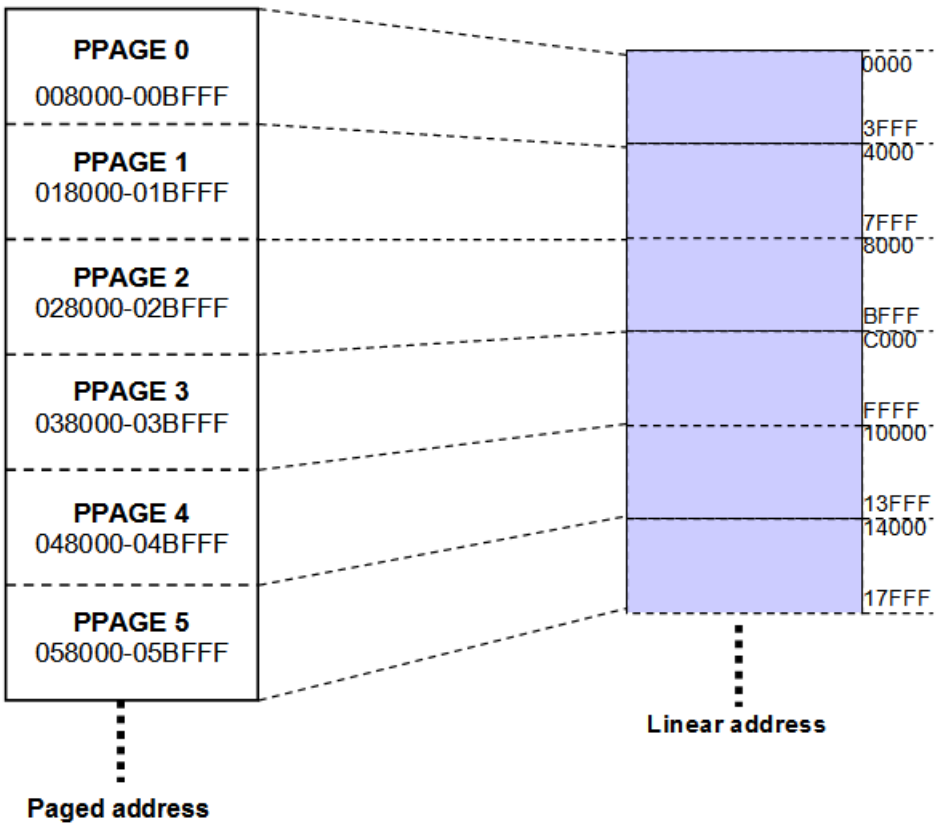

Figure 25-3. Paged/Linear Address Equivalence for MC9S08QE128 Derivative

25.3.2 Compiler Support

CodeWarrior's C compiler predefined a serial macros to manipulate data that to be placed in the extended memory. All these macros are included in `mmu_lda.h`. To define a linear pointer, we should use the special qualifier `__linear` keyword. The keyword __LINEAR_SEG argument is available for the `#pragma CONST_SEG` and `#pragma STRING_SEG`, and tells the compiler to place data into a linear space.

NOTE

No form of pointer dereferencing is supported for `__linear` pointers.

25.3.3 Example

A simple example illustrates placing data in the extended memory, reading a data from the extended memory and storing the data address to a linear pointer.

Listing: Placing Data in the Linear Memory

```
...
#include <mmu_lda.h>

#pragma CONST_SEG __LINEAR_SEG DATA_LINEAR

const int x=2;

#pragma CONST_SEG DEFAULT

int y;

void f() {

  /* y = x; */

  __LOAD_LAP_ADDRESS(x);

  __LOAD_WORD_INC(y);

}
```

In PRM file, place segment `DATA_LINEAR` at the correct address.

NOTE

Objects that are accessed using linear addressing can still be placed in paged segments, but this addressing should be used when objects (for example. variables, arrays) span multiple pages.

NOTE

Make sure not to define the same memory area both as paged and linear.

```
  //PRM file


  SEGMENTS


  ...


  //PPAGE_5     =  READ_ONLY    0x058000 TO 0x05BFFF;
```

```
...

ROM_LINEAR  =  READ_ONLY    0x014000'F TO 0x017FFF'F

...

END

PLACEMENT

...

DEFAULT_ROM, PAGED_ROM

   INTO PPAGE_0,PPAGE_2,PPAGE_4,/*PPAGE_5,*/
PPAGE_6,PPAGE_7,ROM1;

...

DATA_LINEAR     INTO    ROM_LINEAR;

...

END
```

NOTE

When use linear address, a single quote ' (in C '\'') character and a F ('F') character should be added to the end of address to let the linker know it is a linear reference.

Chapter 26
Compiler Messages

This chapter describes the HC(S)08 compiler messages.

NOTE

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

26.1 Compiler Messages

This section lists the compiler messages for HCS08 compiler.

26.1.1 C1: Unknown message occurred

[FATAL]

Description

The application tried to emit a message which was not defined. This is a internal error which should not occur. Please report any occurrences to your support.

Tips

Try to find out the and avoid the reason for the unknown message.

26.1.2 C2: Message overflow, skipping <kind> messages

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The application did show the number of messages of the specific kind as controlled with the options -WmsgNi, -WmsgNw and -WmsgNe. Further options of this kind are not displayed.

Tips

Use the options -WmsgNi, -WmsgNw and -WmsgNe to change the number of messages.

26.1.3 C50: Input file '<file>' not found

[FATAL]

Description

The Application was not able to find a file needed for processing.

Tips

Check if the file really exits. Check if you are using a file name containing spaces (in this case you have to quote it).

26.1.4 C51: Cannot open statistic log file <file>

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

It was not possible to open a statistic output file, therefore no statistics are generated. Note: Not all tools support statistic log files. Even if a tool does not support it, the message still exists, but is never issued in this case.

26.1.5 C52: Error in command line <cmd>

[FATAL]

Description

In case there is an error while processing the command line, this message is issued.

26.1.6 C53: Message <Id> is not used by this version. The mapping of this message is ignored.

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The given message id was not recognized as known message. Usually this message is issued with the options -WmsgS[D|I|W|E]<Num> which should map a specific message to a different message kind.

Example

```
-WmsgSD123456789
```

Tips

There are various reasons why the tool would not recognize a certain message:

- make sure you are using the option with the right tool, say you don't disable linker messages in the compiler preferences
- The message may have existed for an previous version of the tool but was removed for example because a limitation does no longer exist.
- The message was added in a more recent version and the used old version did not support it yet.
- The message did never exist. Maybe a typo?

26.1.7 C54: Option <Option> <Description>.

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

This information is used to inform about special cases for options. One reason this message is used is for options which a previous version did support but this version does no longer support. The message itself contains a descriptive text how to handle this option now.

Tips

Check the manual for all the current option. Check the release notes about the background of this change.

26.1.8 C56: Option value overridden for option <OptionName>. Old value `<OldValue>', new value `<NewValue>'.

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

This message is issued when two or more sub options (of the same option) which are mutually exclusive are passed as arguments to the compiler.

Example

```
crs08.exe -Mb -Ml



/*WARNING C56: Option value overridden for option -M. Old
value 'b', new value 'l'.*/
```

26.1.9 C64: Line Continuation occurred in <FileName>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

In any environment file, the character '\' at the end of a line is taken as line continuation. This line and the next one are handles as one line only. Because the path separation character of MS-DOS is also '\', paths are often incorrectly written ending with '\'. Instead use a '.' after the last '\' to not finish a line with '\' unless you really want a line continuation.

Example

Current Default.env:

```
...
```

```
LIBPATH=c:\Codewarrior\lib\

OBJPATH=c:\Codewarrior\work

...
```

Is taken identical as

```
...

LIBPATH=c:\Codewarrior\libOBJPATH=c:\Codewarrior\work

...
```

Tips

To fix it, append a '.' behind the '\'

```
...

LIBPATH=c:\Codewarrior\lib\.

OBJPATH=c:\Codewarrior\work

...
```

Note Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

26.1.10 C65: Environment macro expansion message '' for <variablename>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

During a environment variable macro substitution an problem did occur. Possible causes are that the named macro did not exist or some length limitation was reached. Also recursive macros may cause this message.

Example

Current variables:

```
...

LIBPATH=${LIBPATH}

...
```

Tips

Check the definition of the environment variable.

26.1.11 C66: Search path <Name> does not exist

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The tool did look for a file which was not found. During the failed search for the file, a non existing path was encountered.

Tips

Check the spelling of your paths. Update the paths when moving a project. Use relative paths.

26.1.12 C1000: Illegal identifier list in declaration

[ERROR]

Description

A function prototype declaration had formal parameter names, but no types were provided for the parameters.

Example

```
int f(i);
```

Tips

Declare the types for the parameters.

26.1.13 C1001: Multiple const declaration makes no sense

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The const qualifier was used more than once for the same variable.

Example

```
const const int i;
```

Tips

Constant variables need only one const qualifier.

See also

- Qualifiers

26.1.14 C1002: Multiple volatile declaration makes no sense

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The volatile qualifier was used more than once for the same variable.

Example

```
volatile volatile int i;
```

Tips

Volatile variables need only one volatile qualifier.

26.1.15 C1003: Illegal combination of qualifiers

[ERROR]

Description

The combination of qualifiers used in this declaration is illegal.

Example

```
int *far near p;
```

Tips

Remove the illegal qualifiers.

26.1.16 C1004: Redefinition of storage class

[ERROR]

Description

A declaration contains more than one storage class.

Example

```
static static int i;
```

Tips

Declare only one storage class per item.

26.1.17 C1005: Illegal storage class

[ERROR]

Description

A declaration contains an illegal storage class.

Example

```
auto int i; // 'auto' illegal for global variables
```

Tips

Apply a correct combination of storage classes.

Seealso

- Storage Classes

26.1.18 C1006: Illegal storage class

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A declaration contains an illegal storage class. This message is used for storage classes which makes no sense and are ignored (e.g. using 'register' for a global variable.

Example

```
register int i; //'register' for global variables
```

Tips

Apply a correct combination of storage classes.

26.1.19 C1007: Type specifier mismatch

[ERROR]

Description

The type declaration is wrong.

Example

```
int float i;
```

Tips

Do not use an illegal type chain.

26.1.20 C1008: Typedef name expected

[ERROR]

Description

A variable or a structure field has to be either one of the standard types (char, int, short, float, ...) or a type declared with a typedef directive.

Example

```
struct A {

  type j; // type is not known

} A;
```

Tips

Use a typedef-name for the object declaration.

26.1.21 C1009: Invalid redeclaration

[ERROR]

Description

Classes, structures and unions may be declared only once. Function redeclarations must have the same parameters and return values as the original declaration. In C++, data objects cannot be redeclared (except with the extern specifier).

Example

```
struct A {

  int i;

};

struct A { // error

  int i;

};
```

Tips

Avoid redeclaration, e.g. guarding include files with ifndef.

26.1.22 C1010: Illegal enum redeclaration

[ERROR]

Description

An enumeration has been declared twice.

Example

```
enum A {

  B

};

enum A { //error

  B

};
```

Tips

Enums have to be declared only once.

26.1.23 C1012: Illegal local function definition

[ERROR]

Description

Non-standard error!

Example

```
void main() {
```

```
  struct A {

    void f() {}

  };

}
```

Tips

The function definitions must always be in the global scope.

```
void main(void) {

  struct A {

    void f();

  };

}

void A::f(void) {

  // function definition in global scope

}
```

26.1.24 C1013: Old style declaration

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The compiler has detected an old style declaration. Old style declarations are common in old non-ANSI sources, however they are accepted. With old style declarations, only the names are in the parameter list and the names and types are declared afterwards.

Example

```
foo(a, b)

  int a, long b;

{

  ...

}
```

Tips

Remove such old style declarations from your application:

```
void foo(int a, long b) {

  ...

}
```

26.1.25 C1014: Integral type expected or enum value out of range

[ERROR]

Description

A non-integral value was assigned to a member of an enum or the enumeration value does not fit into the size specified for the enum (in ANSI-C the enumeration type is int).

Example

```
enum E {


  F="Hello"


};
```

Tips

Enum-members may only get int-values.

26.1.26 C1015: Type is being defined

[ERROR]

Description

The given class or structure was declared as a member of itself. Recursive definition of classes and structures are not allowed.

Example

```
struct A {


  A a;
```

```
};
```

Tips

Use a pointer to the class being defined instead of the class itself.

26.1.27 C1016: Parameter redeclaration not permitted

[ERROR]

Description

A parameter object was declared with the same name as another parameter object of the same function.

Example

```
void f(int i, int i);
```

Tips

Choose another name for the parameter object with the already used name.

26.1.28 C1017: Empty declaration

[ERROR]

Description

A declaration cannot be empty.

Example

```
int;
```

Tips

There must be a name for an object.

26.1.29 C1018: Illegal type composition

[ERROR]

Description

The type was composed with an illegal combination. A typical example is

```
extern struct A dummy[];
```

Example

```
void v[2];
```

Tips

Type compositions must not contain illegal combinations.

26.1.30 C1019: Incompatible type to previous declaration

[ERROR]

Description

The specified identifier was already declared

Example

```
int i;


int i();
```

Tips

Choose another name for the identifier of the second object.

26.1.31 C1020: Incompatible type to previous declaration

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The specified identifier was already declared with different type modifiers. If the option -Ansi is enabled, this warning becomes an error.

Example

```
int i;


int i();
```

Tips

Use the same type modifiers for the first declaration and the redeclaration.

26.1.32 C1021: Bit field type is not 'int'

[ERROR]

Description

Another type than 'int' was used for the bitfield declaration. Some Back Ends may support non-int bitfields, but only if the Compiler switch -Ansi is not given.

Example

```
struct {


  char b:1;


} S;
```

Tips

Use int type for bitfields or remove the -Ansi from the Compiler options.

See also

- C1106: Non-standard bitfield type

26.1.33 C1022: 'far' used in illegal context

[ERROR]

Description

far, rom or uni has been specified for an array parameter where it is not legal to use it. In ANSI C, passing an array to a function always means passing a pointer to the array, because it is not possible to pass an array by value. To indicate that the pointer is a non-standard pointer, non-standard keywords as near or far may be specified if supported.

Example

```
void foo(int far a) {}; // error


void foo(ARRAY far ap) {} // ok: passing a far pointer
```

Tips

Remove the illegal modifier.

26.1.34 C1023: 'near' used in illegal context

[ERROR]

Description

far, rom or uni has been specified for an array parameter where it is not legal to use it. In ANSI C, passing an array to a function always means passing a pointer to the array, because it is not possible to pass an array by value. To indicate that the pointer is a non-standard pointer, non-standard keywords as near or far may be specified if supported.

Example

```
void foo(int near a) {}; // error


void foo(ARRAY near ap) {} // ok: passing a near pointer
```

Tips

Remove the illegal modifier.

26.1.35 C1024: Illegal bit field width

[ERROR]

Description

The type of the bit field is too small for the number of bits specified.

Example

```
struct {


  int b:1234;


} S;
```

Tips

Choose a smaller number of bits, or choose a larger type of the bitfield (if the backend allows such a non-standard extension).

26.1.36 C1025: ',' expected before '...'

[ERROR]

Description

An open parameter list was declared without a ',' before the '...'.

Example

```
void foo(int a ...);
```

Tips

Insert a ',' before the '...'.

26.1.37 C1026: Constant must be initialized

[ERROR]

Description

The specified identifier was declared as const but was not initialized.

Example

```
const int i;         // error


extern const int i;  // ok
```

Tips

Initialize the constant object, or remove the const specifier from the declaration.

26.1.38 C1027: Reference must be initialized

[ERROR]

Description

A reference was not initialized when it was declared.

Example

```
int j;


int&amp; i // = j; missing
```

Tips

Initialize the reference with an object of the same type as the reference points to.

26.1.39 C1028: Member functions cannot be initialized

[ERROR]

Description

A member function of the specified class was initialized.

Tips

Do not initialize member functions in the initialization list for a class or structure.

26.1.40 C1029: Undefined class

[ERROR]

Description

A class is used which is not defined/declared.

Example

```
class A;


class B {


  A::I i;  // error
```

```
};
```

Tips

Define/declare a class before using it.

26.1.41 C1030: Pointer to reference illegal

[ERROR]

Description

A pointer to a reference was declared.

Example

```
void f(int &amp; * p);
```

Tips

The variable must be dereferenced before a pointer to it can be declared.

26.1.42 C1031: Reference to reference illegal

[ERROR]

Description

A reference to a reference was declared.

Tips

This error can be avoided by using pointer syntax and declaring a reference to a pointer.

26.1.43 C1032: Invalid argument expression

[ERROR]

Description

The argument expression of a function call in a Ctor-Init list was illegal.

Example

```
struct A {

  A(int i);

};

struct B : A {

  B();

};

B::B() : A((3) {// error

}
```

Tips

In the argument expression of a Ctor-Init function call, there must be the same number of (as).

26.1.44 C1033: Ident should be base class or data member

[ERROR]

Description

An object in an initialization list was not a base class or member.

Example

```
class A {


int i;


  A(int j) : B(j) {};// error


};
```

Tips

Only a member or base class can be in the initialization list for a class or structure.

26.1.45 C1034: Unknown kind of linkage

[ERROR]

Description

The indicated linkage specifier was not legal. This error is caused by using a linkage specifier that is not supported.

Example

```
extern "MODULA-2" void foo(void);
```

Tips

Only the "C" linkage specifier is supported.

26.1.46 C1035: Friend must be declared in class declaration

[ERROR]

Description

The specified function was declared with the friend specifier outside of a class, structure or union.

Example

```
friend void foo(void);
```

Tips

Do not use the friend specifier outside of class, structure or union declarations.

26.1.47 C1036: Static member functions cannot be virtual

[ERROR]

Description

A static member function was declared as virtual.

Example

```
class A {

  static virtual f(void); // error

};
```

Tips

Do not declare a static member function as virtual.

26.1.48 C1037: Illegal initialization for extern variable in block scope

[ERROR]

Description

A variable with extern storage class cannot be initialized in a function.

Example

```
void f(void) {


  extern int i= 1;


}
```

Tips

Initialize the variable, where it is defined.

26.1.49 C1038: Cannot be friend of myself

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The friend specifier was applied to a function/class inside the scope resolution of the same class.

Example

```
class A {


  friend A::f(); //treated by the compiler as "friend
f();


};
```

Tips

Do not write a scope resolution to the same class for a friend of a class.

26.1.50 C1039: Typedef-name or ClassName expected

[ERROR]

Description

In the current context either a typedef name or a class name was expected.

Example

```
struct A {

  int i;

};

void main() {

  A *a;

  a=new a; // error

}
```

Tips

Write the ident of a type or class/struct tag.

26.1.51 C1040: No valid :: classname specified

[ERROR]

Description

The specified identifier after a scope resolution was not a class, struct, or union.

Example

```
class B {


  class A {


  };


};


class C : B::AA {  // AA is not valid


};
```

Tips

Use an identifier of an already declared class, struct, or union.

26.1.52 C1041: Multiple access specifiers illegal

[ERROR]

Description

The specified base class had more than one access modifier.

Tips

Use only one access modifier (public, private or protected).

26.1.53 C1042: Multiple virtual declaration makes no sense

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The specified class or structure was declared as virtual more than once.

Tips

Use only one virtual modifier for each base class.

26.1.54 C1043: Base class already declared in base list

[ERROR]

Description

The specified class (or structure) appeared more than once in a list of base classes for a derived class.

Example

```
class A {};


class B: A, A {


};
```

Tips

Specify a direct base class only once.

26.1.55 C1044: User defined Constructor is required

[ERROR]

Description

A user-defined constructor should be defined. This error occurs when a constructor should exist, but cannot be generated by the Compiler for the class.

Example

```
class A {


  const int i;


};
```

The compiler can not generate a constructor because he does not know the value for i.

Tips

Define a constructor for the class.

26.1.56 C1045: <Special member function> not generated

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The Compiler option -Cn=Ctr disabled the creation of Compiler generated special member functions (Copy Constructor, Default Constructor, Destructor, Assignment operator).

Tips

If you want the special member functions to be generated by the Compiler, disable the Compiler option -Cn=Ctr.

26.1.57 C1046: Cannot create compiler generated <Special member="" function>=""> for nameless class

[ERROR]

Description

The Compiler could not generate a special member function (Copy Constructor, Default Constructor, Destructor, Assignment operator) for the nameless class.

Example

```
class B {


public:


  B();


};


class {


  B b;


} A;
```

Tips

Give a name to nameless class.

26.1.58 C1047: Local compiler generated <Special member function> not supported

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

Local class declarations would force the compiler to generate local special member functions (Copy Constructor, Default Constructor, Destructor, Assignment operator). But local functions are not supported.

Example

```
;
```

Tips

If you really want the compiler generated special member functions to be created, then declare the class (or struct) in the global scope.

26.1.59 C1048: Generate compiler defined <Special member="" function>=""></Special>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A special member function (Copy Constructor, Default Constructor, Destructor, Assignment operator) was created by the compiler. When a class member or a base class contains a Constructor or a Destructor, then the new class must also have this special function so that the base class Constructor or Destructor is called in every case. If the user does not define one, then the compiler automatically generates one.

Example

```
struct A {

  A(void);

  A(const A&amp;);

  A&amp; operator =(const A&amp; );
```

```
  ~A();


};


struct B : A {


};
```

Tips

If you do not want the compiler to generate the special member functions, then enable the option -Cn=Ctr. The compiler only calls a compiler generated function if it is really necessary. Often a compiler generated function is created, but then never called. Then the smart linker does not waste code space for such functions.

26.1.60 C1049: Members cannot be extern

[ERROR]

Description

Class member cannot have the storage class extern.

Example

```
class A {


  extern int f();


};
```

Tips

Remove the extern specifier.

26.1.61 C1050: Friend must be a class or a function

[ERROR]

Description

The friend specifier can only be used for classes (or structures or unions) and functions.

Example

```
typedef int I;


struct A {


  friend I;  // illegal


};
```

Tips

Use the friend specifier only for classes (or structures or unions) and functions.

26.1.62 C1051: Invalid function body

[ERROR]

Description

The function body of a member function inside a class declaration is invalid.

Example

```
struct A {
```

```
  void f() { {{int i; }

};
```

Tips

The function body of a member function inside a class declaration must have the same number of "{" as "}".

26.1.63 C1052: Unions cannot have class/struct object members containing Con/Destructor/Assign-Operator

[ERROR]

Description

The specified union member was declared with a special member (Con/Destructor/ Assign-Operator).

Example

```
class A {

  A(void);

};

union B {

  A a;

};
```

Tips

The union member may contain only compiler generated special members. So try to compile with the option -Cn=Ctr enabled.

26.1.64 C1053: Nameless class cannot have member functions

[ERROR]

Description

A function was declared in a nameless class.

Example

```
class {


  void f(void);


};
```

Tips

Name the nameless class, or remove all member functions of the nameless class.

26.1.65 C1054: Incomplete type or function in class/struct/union

[ERROR]

Description

A used type in a function, class, struct or union was not completely defined.

Tips

Define types before using them.

26.1.66 C1055: External linkage for class members not possible

[ERROR]

Description

Member redeclarations cannot have external linkage.

Example

```
struct A {

  f();

}a;

extern "C" A::f() {return 3;}
```

Tips

Do not declare members as extern.

26.1.67 C1056: Friend specifier is illegal for data declarations

[ERROR]

Description

The friend specifier cannot be used for data declarations.

Example

```
class A {

  friend int a;
```

```
};
```

Tips

Remove the friend specifier from the data declaration.

26.1.68 C1057: Wrong return type for <FunctionKind>

[ERROR]

Description

The declaration of a function of FunctionKind contained an illegal return type.

Tips

Depending on FunctionKind:

- operator -> must return a pointer or a reference or an instance of a class, structure or union
- operator delete must return void
- operator new must return void *

26.1.69 C1058: Return type for FunctionKind must be <ReturnType>

[ERROR]

Description

Some special functions must have certain return types. An occurred function did have an illegal return type.

Tips

Depending on FunctionKind:

- operator -> must return a pointer or a reference or an instance of a class, structure or union
- operator delete must return void
- operator new must return void *

26.1.70 C1059: Parameter type for <FunctionKind> parameter <No> must be <Type>

[ERROR]

Description

The declaration of a function of FunctionKind has a parameter of a wrong type.

Tips

Depending on FunctionKind:

- operator new parameter 1 must be unsigned int
- operator delete parameter 1 must be void *
- operator delete parameter 2 must be unsigned int

26.1.71 C1060: <FunctionKind> wrong number of parameters

[ERROR]

Description

The declaration of a function of FunctionKind has a wrong number of parameters

Tips

Depending on FunctionKind: member operator delete must have one or two parameters

26.1.72 C1061: Conversion operator must not have return type specified before operator keyword

[ERROR]

Description

A user-defined conversion cannot specify a return type.

Example

```
class A {

  public:

      int operator int() {return value;}// error

    operator int() {return value;}    // ok

  private:

    int value;

}
```

Tips

Do not specify a return type before the operator keyword.

26.1.73 C1062: Delete can only be global, if parameter is (void *)

[ERROR]

Description

Global declarations/definitions of operator delete are allowed to have only one parameter.

Tips

Declare only one parameter for the global delete operator. Or declare the delete operator as a class member, where it can have 2 parameters.

26.1.74 C1063: Global or static-member operators must have a class as first parameter

[ERROR]

Description

The specified overloaded operator was declared without a class parameter.

Example

```
int operator+ (int, int); // error;
```

Tips

The first parameter must be of class type.

26.1.75 C1064: Constructor must not have return type

[ERROR]

Description

The specified constructor returned a value, or the class name is used for a member.

Example

```
struct C {

  int C();  // error

  C();      // ok

};
```

Tips

Do not declare a return type for constructors.

26.1.76 C1065: 'inline' is the only legal storage class for Constructors

[ERROR]

Description

The specified constructor has an illegal storage class (auto, register, static, extern, virtual).

Tips

The only possible storage class for constructors is inline.

26.1.77 C1066: Destructor must not have return type

[ERROR]

Description

The specified destructor returned a value.

Tips

Do not declare a return type for destructors.

26.1.78 C1067: Object is missing decl specifiers

[ERROR]

Description

An object was declared without decl-specifiers (type, modifiers, ...). There is no error, if compiling C-source without the -ANSI option.

Example

```
i
```

Tips

Apply decl-specifiers for the object, or compile without the options -ANSI and -C++.

26.1.79 C1068: Illegal storage class for Destructor

[ERROR]

Description

The specified destructor has an illegal storage class (static, extern).

Tips

Do not use the storage classes static and extern for destructors.

26.1.80 C1069: Wrong use of far/near/rom/uni/paged in local scope

[ERROR]

Description

The far/near/rom/uni/paged keyword has no effect in the local declaration of the given identifier. far may be used to indicate a special addressing mode for a global variable only. Note that such additional keywords are not ANSI compliant and not supported on all targets.

Example

```
far int i;  // legal on some targets


void foo(void) {


  far int j;  // error message C1069
```

```
}
```

Tips

Remove the far/near/rom/uni/paged qualifier, or declare the object in the global scope.

26.1.81 C1070: Object of incomplete type

[ERROR]

Description

An Object with an undefined or not completely defined type was used.

Example

```
void f(struct A a) {


}
```

Tips

Check the spelling of the usage and the definition of this type. It is legal in C to pass a pointer to a undefined structure, so examine if is possible to pass a pointer to this type rather than the value itself.

26.1.82 C1071: Redefined extern to static

[ERROR]

Description

An extern identifier was redefined as static.

Example

```
extern int i;
```

```
static int i;  // error
```

Tips

Remove either the extern specifier of the first declaration, or the static specifier of the second occurrence of the identifier.

26.1.83 C1072: Redefined extern to static

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

If the option -ANSI is disabled, the nonstandard extension issues only a warning, not an error.

Example

```
extern int i;


static int i;  // warning
```

Tips

Remove either the extern specifier of the first declaration, or the static specifier of the second occurrence of the identifier.

26.1.84 C1073: Linkage specification contradicts earlier specification

[ERROR]

Description

The specified function was already declared with a different linkage specifier. This error can be caused by different linkage specifiers found in include files.

Example

```
int f(int i);


extern "C" int f(int i);
```

Tips

Use the same linkage specification for the same function/variable.

26.1.85 C1074: Wrong member function definition

[ERROR]

Description

The specified member function was not declared in the class/structure for the given parameters.

Example

```
class A {


  void f(int i);


};


void A::f(int i, int i) {  // error


}
```

Tips

Check the parameter lists of the member function declarations in the class declaration and the member function declarations/definitions outside the class declaration.

26.1.86 C1075: Typedef object id already used as tag

[ERROR]

Description

The identifier was already used as tag. In C++, tags have the same namespace than objects. So there would be no name conflict compiling in C.

Example

```
typedef const struct A A; // error in C++, ANSI-C ok
```

Tips

Compile without the option -C++, or choose another name for the typedef object id.

26.1.87 C1076: Illegal scope resolution in member declaration

[ERROR]

Description

An access declaration was made for the specified identifier, but it is not a member of a base class.

Example

```
struct A {

  int i;

};

struct B {
```

```
  int j;


};


struct C : A {


  A::i; // ok


  B::j; // error


};
```

Tips

Put the owner class of the specified member into the base class list, or do without the access declaration.

26.1.88 C1077: <FunctionKind> must not have parameters

[ERROR]

Description

Parameters were declared where it is illegal.

Tips

Do not declare parameters for Destructors and Conversions.

26.1.89 C1078: <FunctionKind> must be a function

[ERROR]

Description

A constructor, destructor, operator or conversion operator was declared as a variable.

Example

```
struct A {

  int A;

};
```

Tips

Constructors, destructors, operators and conversion operators must be declared as functions.

26.1.90 C1080: Constructor/destructor: Parenthesis missing

[ERROR]

Description

A redeclaration of a constructor/destructor is done without parenthesis.

Example

```
struct A {

  ~A();

};

A::~A;   // error
```

```
A::~A(); // ok
```

Tips

Add parenthesis to the redeclaration of the constructor/destructor.

26.1.91 C1081: Not a static member

[ERROR]

Description

The specified identifier was not a static member of a class or structure.

Example

```
struct A {

  int i;

};

void main() {

  A::i=4;   // error

}
```

Tips

Use a member access operator (. or ->) with a class or structure object; or declare the member as static.

26.1.92 C1082: <FunctionKind> must be non-static member of a class/struct

[ERROR]

Description

The specified overloaded operator was not a member of a class, structure or union, and/or was declared as static. FunctionKind can be a conversion or an operator =, -> or ().

Tips

Declare the function inside a class declaration without the static storage class.

26.1.93 C1084: Not a member

[ERROR]

Description

An ident has been used which is not a member of a class or a struct field.

Tips

Check the struct/class declaration.

26.1.94 C1085: <ident> is not a member

[ERROR]

Description

A nonmember of a structure or union was incorrectly used.

Example

```
struct A {


  int i;
```

```
};

void main() {

  A a;

  a.r=3;  // error

}
```

Tips

Using . or ->, specify a declared member.

26.1.95 C1086: Global unary operator must have one parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global member unary operator must have exactly one parameter.

26.1.96 C1087: Static unary operator must have one parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static member unary operator must have exactly one parameter.

26.1.97 C1088: Unary operator must have no parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Member unary operator must have no parameters.

26.1.98 C1089: Global binary operator must have two parameters

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global binary operator must have two parameters.

26.1.99 C1090: Static binary operator must have two parameters

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static binary operator must have two parameters.

26.1.100 C1091: Binary operator must have one parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Binary operator must have one parameter.

26.1.101 C1092: Global unary/binary operator must have one or two parameters

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global unary/binary operator must have one or two parameters.

26.1.102 C1093: Static unary/binary operator must have one or two parameters

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static unary/binary operator must have one or two parameters.

26.1.103 C1094: Unary/binary operator must have no or one parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Unary/binary operator must have no or one parameter.

26.1.104 C1095: Postfix ++/-- operator must have integer parameter

[ERROR]

Description

The specified overloaded operator was incorrectly declared with the wrong type of parameters.

Tips

Postfix ++/-- operator must have integer parameter.

26.1.105 C1096: Illegal index value

[ERROR]

Description

The index value of an array declaration was equal or less than 0.

Example

```
int i[0]; // error;


// for 16bit int this is 0x8CA0 or -29536 !


static char dct_data[400*90];
```

Tips

The index value must be greater than 0. If the index value is a calculated one, use a 'u' to make the calculation unsigned (e.g. 400*90u).

26.1.106 C1097: Array bounds missing

[ERROR]

Description

The non-first dimension of an array has no subscript value.

Example

```
int i[3][]; // error


int j[][4]; // ok
```

Tips

Specify a subscript value for the non-first dimensions of an array.

26.1.107 C1098: Modifiers for non-member or static member functions illegal

[ERROR]

Description

The specified non-member function was declared with a modifier.

Example

```
void f() const; // error;
```

Tips

Do not use modifiers on non-member or static member functions.

26.1.108 C1099: Not a parameter type

[ERROR]

Description

An illegal type specification was parsed.

Example

```
struct A {

  int i;

};

void f(A::i); // error
```

Tips

Specify a correct type for the parameter.

26.1.109 C1100: Reference to void illegal

[ERROR]

Description

The specified identifier was declared as a reference to void.

Example

```
void &amp;vr;  // error;
```

Tips

Don not declare references to a void type.

26.1.110 C1101: Reference to bitfield illegal

[ERROR]

Description

A reference to the specified bit field was declared.

Example

```
struct A {


  int &amp;i : 3; // error


};
```

Tips

Do not declare references to bitfields.

26.1.111 C1102: Array of reference illegal

[ERROR]

Description

A reference the specified array was declared.

Example

```
extern int &amp;j[20];
```

Tips

Do not declare references to arrays.

26.1.112 C1103: Second C linkage of overloaded function not allowed

[ERROR]

Description

More than one overloaded function was declared with C linkage.

Example

```
extern "C" void f(int);


extern "C" void f(long);
```

Tips

When using C linkage, only one form of a given function can be made external.

26.1.113 C1104: Bit field type is neither integral nor enum type

[ERROR]

Description

Bit fields must have an integral type (or enum type for C).

Example

```
struct A {


  double d:1;


};
```

Tips

Specify an integral type (int, long, short, ...) instead of the non-integral type.

26.1.114 C1105: Backend does not support non-int bitfields

[ERROR]

Description

Bit fields must be of integer type. Any other integral or non-integral type is illegal. Some backends support non int bitfields, others do not. See the chapter backend for details.

Example

```
struct A {


  char i:2;


}


When the actual backend supports non-int bitfields, this
message does not occur.
```

Tips

Specify an integer-type (int, signed int or unsigned int) for the bitfield.

26.1.115 C1106: Non-standard bitfield type

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

Some of the Back Ends allow bitfield structure members of any integral type.

Example

```
struct bitfields {

  unsigned short j:4; // warning

};
```

Tips

If you want portable code, use an integer-type for bitfields.

26.1.116 C1107: Long long bit fields not supported yet

[ERROR]

Description

Long long bit fields are not yet supported.

Example

```
struct A {

  long long l:64;
```

```
};
```

Tips

Do not use long long bit fields.

26.1.117 C1108: Constructor cannot have own class/struct type as first and only parameter

[ERROR]

Description

A constructor was declared with one parameter, and the parameter has the type of the class itself.

Example

```
struct B {


  B(B b); // error


};
```

Tips

Use a reference of the class instead of the class itself.

26.1.118 C1109: Generate call to Copy Constructor

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

An instance of a class was passed as argument of a function, needing to be copied onto the stack w ith the Copy Constructor. Or an instance of a class was used as initializer of another instance of the same class, needing to be copied to the new class instance with the Copy Constructor

Example

```
struct A {

  A(A &amp;amp;);

  A();

};

void f(A a);

void main(void) {

  A a;

  f(a); // generate call to copy ctor

}
```

Tips

If conventional structure copying is desired, try to compile with the option -Cn=Ctr and do not declare copy constructors manually.

26.1.119 C1110: Inline specifier is illegal for data declarations

[ERROR]

Description

The inline specifier was used for a data declaration.

Example

```
inline int i;
```

Tips

Do not use inline specifiers for data declarations.

26.1.120 C1111: Bitfield cannot have indirection

[ERROR]

Description

A bitfield declaration must not contain a *. There are no pointer to bits in C. Use instead a pointer to the structure containing the bitfield.

Example

```
struct bitfield {


  int *i : 2;  // error


};
```

Tips

Do not use pointers in bitfield declarations.

26.1.121 C1112: Interrupt specifier is illegal for data declaration

[ERROR]

Description

The interrupt specifier was applied to a data declaration.

Example

```
interrupt int i;  // error
```

Tips

Apply the interrupt specifier for functions only

26.1.122 C1113: Interrupt specifier used twice for same function

[ERROR]

Description

The interrupt specifier was used twice for the same function.

Example

```
interrupt 4 void interrupt 2 f(); // error
```

Tips

26.1.123 C1114: Illegal interrupt number

[ERROR]

Description

The specified vector entry number for the interrupt was illegal.

Example

```
interrupt 1000 void f(); // error
```

Tips

Check the backend for the range of legal interrupt numbers! The interrupt number is not the same as the address of the interrupt vector table entry. The mapping from the interrupt number to the vector address is backend specific.

26.1.124 C1115: Template declaration must be class or function

[ERROR]

Description

A non-class/non-function was specified in a template declaration.

Example

```
template<class A> int i; // error
```

Tips

Template declarations must be class or function

26.1.125 C1116: Template class needs a tag

[ERROR]

Description

A template class was specified without a tag.

Example

```
template<class A> struct {  // error


  A a;


};
```

Tips

Use tags for template classes.

26.1.126 C1117: Illegal template/non-template redeclaration

[ERROR]

Description

An illegal template/non-template redeclaration was found.

Example

```
template<class A> struct B {

 A a;

};

struct B {  // error

 A a;

};
```

Tips

Correct the source. Protect header files with from multiple inclusion.

26.1.127 C1118: Only bases and class member functions can be virtual

[ERROR]

Description

Because virtual functions are called only for objects of class types, you cannot declare global functions or union member functions as 'virtual'.

Example

```
virtual void f(void); // ERROR: definition of a global

                      // virtual function.

union U {

  virtual void f(void); // ERROR: virtual union member

                        // function

};
```

Tips

Do not declare a global function or a union member function as virtual.

26.1.128 C1119: Pure virtual function qualifier should be (=0)

[ERROR]

Description

The '=0' qualifier is used to declare a pure virtual function. Following example shows an ill-formed declaration.

Example

```
class A{     // class


 public:


  virtual void f(void)=2; // ill-formed pure virtual


                          // function declaration.


};
```

Tips

Correct the source.

26.1.129 C1120: Only virtual functions can be pure

[ERROR]

Description

Only a virtual function can be declared as pure. Following example shows an hill-formed declaration.

Example

```
class A{            // class


  public:


  void f(void)=0;   // ill-formed declaration.


};
```

Tips

Make the function virtual. For overloaded functions check if the parameters are identical to the base virtual function.

26.1.130 C1121: Definition needed if called with explicit scope resolution

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A definition for a pure virtual function is not needed unless explicitly called with the qualified-id syntax (nested-name-specifier [template] unqualified-id).

Example

```
class A{

 public:

  virtual void f(void) = 0;  // pure virtual function.

};

class B : public A{

 public:

  void f(void){ int local=0; }

};
```

```
void main(void){

  B b;

  b.A::f();               // generate a linking error cause

                          // no object is defined.

  b.f();                  // call the function defined in

                          // B class.

}
```

Tips

Correct the source.

26.1.131 C1122: Cannot instantiate abstract class object

[ERROR]

Description

An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as objects representing a base class of a class derived from it.

Example

```
class A{
```

```
public:

  virtual void f(void) = 0; //pure virtual function ==> A

                            //is an abstract class

};

void main(void){

  A a;

}
```

Tips

Use a pointer/reference to the object:

```
void main(void){

  A *pa;

}
```

Use a derived class from abstract class:

```
class B : public A{

public:
```

```
  void f(void){}


};


void main(void){


  B b;


}
```

26.1.132 C1123: Cannot instantiate abstract class as argument type

[ERROR]

Description

An abstract class may not be used as argument type

Example

```
class A{


public:


  virtual void f(void) = 0; // pure virtual function


                            // ==> A is an abstract class


};
```

```
void main(void){

  void fct(A);

}
```

Tips

Use a pointer/reference to the object:

```
void main(void){

  void fct(A *);

}
```

Use a derived class from abstract class

```
class B : public A{

public:

  void f(void){}

};

void main(void){

  void fct(B);
```

```
}
```

26.1.133 C1124: Cannot instantiate abstract class as return type

[ERROR]

Description

An abstract class may not be used as a function return type.

Example

```
class A{

public:

  virtual void f(void) = 0; //pure virtual function ==> A

                            //is an abstract class

};

void main(void){

  A fct(void);

}
```

Tips

Use a pointer/reference to the object

```
void main(void){

  A *fct(void);

}
```

Use a derived class from abstract class

```
class B : public A{

public:

  void f(void){}

};

void main(void){

  B fct(void);

}
```

26.1.134 C1125: Cannot instantiate abstract class as a type of explicit conversion

[ERROR]

Description

An abstract class may not be used as a type of an explicit conversion.

Example

```
class A{


public:


  virtual void f(void) = 0; //pure virtual function ==>
A is an abstract class


};


class B : public A{


public:


  void f(void){}


};


void main(void){


  A *pa;


  B b;









  pa = &amp;(A)b;
```

```
}
```

Tips

Use a pointer/reference to the object

```
void main(void){

  A *pa;

  B b;



  pa = (A *)b;

}
```

26.1.135 C1126: Abstract class cause inheriting pure virtual without overriding function(s)

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

Pure virtual functions are inherited as pure virtual functions.

Example

```
class A{

public:
```

```
  virtual void f(void) = 0;

  virtual void g(void) = 0;

};

class B : public A{

public:

  void f(void){}

  // void B::g(void) is inherited pure virtual

  // ==> B is an implicit abstract class

};
```

26.1.136 C1127: Constant void type probably makes no sense

[INFORMATION]

Description

A pointer/reference to a constant void type was declared

Example

```
const void *cvp;   // warning (pointer to constant void)
```

```
void *const vpc;  // no warning (constant pointer)
```

Tips

A pointer to a constant type is a different thing than a constant pointer.

26.1.137 C1128: Class contains private members only

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A class was declared with only private members.

Example

```
class A {


 private:


   int i;


};
```

Tips

You can never access any member of a class containing only private members!

26.1.138 C1129: Parameter list missing in pointer to member function type

[ERROR]

Description

The current declaration is neither the one of pointer to member nor the one of pointer to member function. But something looking like melting of both.

Example

```
class A{


public:


  int a;


};


typedef int (A::*pm);
```

Tips

Use the standard declaration: for a pointer to member 'type class::*ident' and for a pointer to member function 'type (class::*ident)(param list)'

```
class A{


public:


  int a;


  void fct(void);


};


typedef int A::*pmi;
```

```
typedef void (A::*pmf)(void);
```

26.1.139 C1130: This C++ feature is disabled in your current cC++/EC++ configuration

[ERROR]

Description

Either the Embedded C++ language (EC++) is enabled (option -C++e), or the compactC++ language (cC++) is enabled (option -C++c) plus the appropriate feature is disabled (option -Cn). Following features could be disabled: virtual functions templates pointer to member multiple inheritance and virtual base classes class parameters and class returns

Tips

If you really don't want to use this C++ feature, you have to find a workaround to the problem. Otherwise change the C++ language configuration with the options -C++ and -Cn, or use the advanced option dialog

26.1.140 C1131: Illegal use of global variable address modifier

[ERROR]

Description

The global variable address modifier was used for another object than a global variable.

Example

```
int glob @0xf90b;  // ok, the global variable "glob" is
at 0xf90b



int *globp @0xf80b = &amp;glob; // ok, the global
variable
```

```
                                // "globp" is at 0xf80b and

                                // points to "glob"

void g() @0x40c0;  // error (the object is a function)

void f() {

  int i @0x40cc; // error (the object is a local variable)

}
```

Tips

Global variable address modifiers can only be used for global variables. They cannot be used for functions or local variables. Global variable address modifiers are a not ANSI standard. So the option -Ansi has to be disabled. To Put a function at a fixed address, use a pragma CODE_SEG to specify a segment for a function. Then map the function in the prm file with the linker. To call a function at a absolute address use a cast:

```
#define f ((void (*) (void)) 0x100)

void main(void) {

 f();

}
```

26.1.141 C1132: Cannot define an anonymous type inside parentheses

[ERROR]

Description

An anonymous type was defined inside parentheses. This is illegal in C++.

Example

```
void f(enum {BB,AA} a) { //  C ok, C++ error


}
```

Tips

Define the type in the global scope.

26.1.142 C1133: Such an initialization requires STATIC CONST INTEGRAL member

[ERROR]

Description

A static CONST member of integral type may be initialized by a constant expression in its member declaration within the class declaration.

Example

```
int e = 0;


class A{


public:
```

```
  static int a = 1; // ERROR: non-const initialized

  const int b = 2; // ERROR: non-static initialized

  static const float c = 3.0;// ERROR: non-integral

                             // initializer

static const int d = e; // ERROR: non-const initializer

  // ...

};
```

Tips

```
class A{

public:

  static const int a = 5; // Initialization

  // ...

};

const int A::a;             // Definition
```

or the other way round:

```
class A{

public:

  static const int a;     // Definition

  // ...

};

const int A::a = 5;         // Initialization
```

26.1.143 C1134: Static data members are not allowed in local classes

[ERROR]

Description

Static members of a local class have no linkage and cannot be defined outside the class declaration. It follows that a local class cannot have static data members.

Example

```
void foo(void){

  class A{

  public:
```

```
    static int a; // ERROR because static data member

    static void myFct(void); // OK because static method

  };

}
```

Tips

Remove the static specifier from the data member declarations of any local class.

```
void foo(void){

  class A{

  public:

    int a; // OK because data member.

    static void myFct(void); // OK because static method

  };

}
```

26.1.144 C1135: Ignore Storage Class Specifier cause it only applies on objects

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The specified Storage Class Specifier is not taken in account by the compiler, because it does not apply to an object.

Example

```
static class A{

  public:

  int a;

};
```

Tips

Remove the Storage Class Specifier from the class definition and apply it to the instances.

```
class A{

  public:

  int a;

};

static A myClassA;
```

26.1.145 C1136: Class <Ident> is not a correct nested class of class <Ident>

[ERROR]

Description

Error detected while parsing the scope resolution of a nested class.

Example

```
class A{

  class B;

}

class A::C{};
```

Tips

Check that the scope resolution matches the nested class.

```
class A{

  class B;

}

class A::B{};
```

26.1.146 C1137: Unknown or illegal segment name

[ERROR]

Description

A segment name used in a segment specifier was not defined with a segment pragma before.

Example

```
#pragma DATA_SEG AA


#pragma DATA_SEG DEFAULT


int i @ "AA"; // OK


int i @ "BB"; // Error C1137, segment name BB not known
```

Tips

All segment names must be known before they are used to avoid tipping mistakes and to specify a place to put segment modifiers.

See also

- pragma DATA_SEG
- pragma CONST_SEG
- pragma CODE_SEG

26.1.147 C1138: Illegal segment type

[ERROR]

Description

A segment name with an illegal type was specified. Functions can only be placed into segments of type CODE_SEG, variable/constants can only be placed into segments of type DATA_SEG or CONST_SEG.

Example

```
#pragma DATA_SEG AA


#pragma CODE_SEG BB


int i @ "AA"; // OK


int i @ "BB"; // Error C1138, data cannot be placed in
codeseg
```

Tips

Use different segment names for data and code. To place constants into rom, use segments of type CONST_SEG.

See also

- pragma DATA_SEG
- pragma CONST_SEG
- pragma CODE_SEG

26.1.148 C1139: Interrupt routine should not have any return value nor any parameter

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

Interrupt routines should not have any return value nor any parameter. In C++, member functions cannot be interrupt routines due to hidden THIS parameter. Another problem may be that a pragma TRAP_PROC is active where it should not be.

Example

```
int interrupt myFct1(void){


  return 4;
```

```
}

#pragma TRAP_PROC

void myFct2(int param){ }

class A{

public:

  void myFctMbr(void);

};

void interrupt A::myFctMbr(void){}
```

Tips

Remove all return value and all parameter (even hidden, e.g. 'this' pointer for C++):

```
void interrupt myFct1(void){ }

#pragma TRAP_PROC

void myFct2(void){

}
```

```
class A{


public:


  void myFctMbr(void);


};


void A::myFctMbr(void){ }


void interrupt myInterFct(void){ }
```

26.1.149 C1140: This function is already declared and has a different prototype

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

There are several different prototypes for the same function in a C module.

Example

```
int Foo (char,float,int,int* );


int Foo (char,float,int,int**);


int Foo (char,char,int,int**);
```

Tips

Check which one is correct and remove the other(s).

26.1.150 C1141: Ident <ident> cannot be allocated in global register

[ERROR]

Description

The global variable 'ident' cannot be allocated in the specified register. There are two possible reasons: The type of the variable is not supported to be accessed in a register. or The specified register number is not possible (e.g. used for parameter passing).

Example

```
extern int glob_var @__REG 4; //r4 is used for parameters
```

Tips

Consider the ABI of the target processor.

26.1.151 C1142: Invalid Cosmic modifier. Accepted: , , or (-ANSI off)

[ERROR]

Description

The modified after the @ was not recognized.

Example

```
@nearer unsigned char index;
```

Tips

Check the spelling. Cosmic modifiers are only supported with the option -Ccx. Not all backends do support all qualifiers. Consider using a pragma DATA_SEG with a qualifier instead.

26.1.152 C1143: Ambiguous Cosmic space modifier. Only one per declaration allowed

[ERROR]

Description

Multiple cosmic modifiers where found for a single declaration. Use only one of them.

Example

```
@near @far unsigned char index;
```

Tips

Cosmic modifiers are only supported with the option -Ccx. Not all backends do support all qualifiers. Only use one qualifier. Consider using a pragma DATA_SEG with a qualifier instead.

26.1.153 C1144: Multiple restrict declaration makes no sense

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The restrict qualifier should only be applied once and not several times.

Example

```
int * restrict restrict pointer;
```

Tips

Only specify restrict once.

26.1.154 C1390: Implicit virtual function

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The 'virtual' keyword is needed only in the base class's declaration of the function; any subsequent declarations in derived classes are virtual by default.

Example

```
class A{ //base class

 public:

  virtual void f(void){ glob=2; } //definition of a

                                  //virtual function

};

class B : public A{ //derived class

 public:

  void f(void){ glob=3; } //overriding function

                          //IMPLICIT VIRTUAL FUNCTION

};
```

Example2:

```
class A{ // base class
```

```
 public:

  virtual void f(void){ glob=2; } //definition of a

                                  //virtual function.

};

class B : public A{ //derived class

 public:

  virtual void f(void){ glob=3; } //overriding function:

                                  //'virtual' is not

                                  // necessary here

};
```

Example3:

```
class A{ //base class

 public:

  virtual void f(void){ glob=2; } //definition of a
```

```
                                    //virtual function.

};

class B : public A { //derived class

 public:

  void f(int a){ glob=3+a; } // not overriding function

};
```

Tips

A derived class's version of a virtual function must have the same parameter list and return type as those of the base class. If these are different, the function is not considered a redefinition of the virtual function. A redefined virtual function cannot differ from the original only by the return type.

26.1.155 C1391: Pseudo Base Class is added to this class

[DISABLE, **INFORMATION** ,WARNING,ERROR]

Description

The virtual keyword ensures that only one copy of the subobject is included in the memory scope of the object. This single copy is the PSEUDO BASE CLASS.

Example

```
class A{ //base class

    // member list
```

```
};

class B : public virtual A { //derived class

 public:

  // member list

};

class C : public virtual A { //derived class

 public:

  // member list

};

class D : public B, public C { //derived class

 public:

  // member list

};
```

Tips

According to this definition, an object 'd' would have the following memory layout:

```
A part

B part

------

A part

C part

------

D part
```

But the 'virtual' keyword makes the compiler to generate the following memory layout.

```
B part

------

C part

------

A part     // This is the PSEUDO BASE CLASS of class D
```

```
------


D part
```

In addition, a pointer to the PSEUDO BASE CLASS is included in each base class that previously derived from virtual base class (here B and C classes).

26.1.156 C1392: Pointer to virtual methods table not qualified for code address space (use -Qvtprom or -Qvtpuni)

[ERROR]

Description

If a virtual methods table of a class is forced to be allocated in code address space (only possible with Harvard architecture), the pointers to the virtual methods table must be qualified with a 'rom' attribute (i.e. rom pointer). This message currently appears only if you specify the compiler option -Cc (allocate 'const' objects in rom). For Harvard targets all virtual methods tables are put into code address space because the virtual methods tables are always constant. In this case the compiler generated pointers to the virtual methods table must be qualified with the rom or uni attribute (see under Tips).

Tips

Qualify virtual table pointers with 'rom' by setting the compiler option -Qvtprom.

See also

- Option -Qvtp

26.1.157 C1393: Delta value does not fit into range (option -Tvtd)

[ERROR]

Description

An option ' -Tvtd' is provided by the compiler to let the user specify the delta value size. The delta value is used in virtual functions management in order to set up the value of the THIS pointer. Delta value is stored in virtual tables. Letting the user specify the size of the delta value can save memory space. But one can imagine that the specified size can be too short, that is the aim of this error message.

Example

```
class A{

public:

  long a[33]

};

class B{

public:

  void virtual fct2 (void){}

};

class C : public A, public B{

public:

};
```

```
void main (void){


  C c;


  c.fct2();


}


If the previous example is compiled using the option \c
-Tvtd1 then the minimal value allowed for delta is (-128)
and a real value of delta is -(4*33)=-132.
```

Tips

Specify a larger size for the delta value: -Tvtd2.

See also

- option -T
- C++ Front End

26.1.158 C1395: Classes should be the same or derive one from another

[ERROR]

Description

Pointer to member is defined to point on a class member, then classes (the one which member belongs to and the one where the pointer to member points) have to be identical. If the member is inherited from a base class then classes can be different.

Example

```
class A{
```

```
public:

  int a;

  void fct1(void){}

};

class B{

public:

  int b;

  void fct2(void){}

};

void main(void){

  int B::*pmi = &amp;A::a;

  void (B::*pmf)() = &amp;A::fct1;

}
```

Tips

Use the same classes

```
class A{

public:

  int a;

  void fct1(void){}

};

class B{

public:

  int b;

  void fct2(void){}

};

void main(void){

  int A::*pmi = &amp;A::a;

  void (A::*pmf)() = &amp;A::fct1;
```

```
}
```

Use classes which derive one from an other

```
class A{

public:

  int a;

  void fct1(void){}

};

class B : public A{

public:

  int b;

  void fct2(void){}

};

void main(void){

  int B::*pmi = &amp;A::a;
```

```
  void (B::*pmf)() = &amp;A::fct1;


}
```

26.1.159 C1396: No pointer to STATIC member: use classic pointer

[ERROR]

Description

Syntax of pointer to member cannot be used to point to a static member. Static member have to be pointed in the classic way.

Example

```
int glob;


class A{


public:


  static int a;


  static void fct(void){}


};


void main(void){
```

```
  int A::*pmi = &amp;A::a;

  void (A::*pmf)() = &amp;A::fct;

}
```

Tips

Use the classic pointer to point static members

```
class A{

public:

  static int a;

  static void fct(void){}

};

void main(void){

  A aClass;

  int *pmi = &amp;aClass.a;

  void (*pmf)() = &amp;aClass.fct;
```

```
}
```

26.1.160 C1397: Kind of member and kind of pointer to member are not compatible

[ERROR]

Description

A pointer to member can not point to a function member and a pointer to function member can not point a member.

Example

```
class A{

public:

  int b;

  int c;

  void fct(){}

  void fct2(){}

};

void main(void){
```

```
  int A::*pmi = &amp;A::b;

  void (A::* pmf)() = &amp;A::fct;

  pmi=&amp;A::fct2;

  pmf=&amp;A::c;

}
```

Tips

```
class A{

public:

  int b;

  int c;

  void fct(){}

  void fct2(){}

};

void main(void){
```

```
    int A::*pmi = &amp;A::b;


    void (A::* pmf)() = &amp;A::fct;


    pmf=&amp;A::fct2;


    pmi=&amp;A::c;


  }
```

26.1.161 C1398: Pointer to member offset does not fit into range of given type (option -Tpmo)

[ERROR]

Description

An option -Tpmo is provided by the compiler to let the user specify the pointer to member offset value size. Letting the user specify the size of the offset value can save memory space. But one can imagine that the specified size is too short, that is the aim of this message.

Example

```
  class A{


  public:


    long a[33];


    int b;
```

```
};

void main (void){

  A myA;

  int A::*pmi;



  pmi = &amp;A::b;

  myA.*pmi = 5;

}

If the previous example is compiled using the option  \c
-Tpmo1 then the maximal value allowed for offset is (127)
and a real value of offset is (4*33)=132.
```

Tips

Specify a larger size for the offset value: -Tpmo2.

See also

- option -T
- C++ Front End.

26.1.162 C1400: Missing parameter name in function head

[ERROR]

Description

There was no identifier for a name of the parameter. Only the type was specified. In function declarations, this is legal. But in function definitions, it's illegal.

Example

```
void f(int) {}  // error


void f(int);    // ok
```

Tips

Declare a name for the parameter. In C++ parameter names must not be specified.

26.1.163 C1401: This C++ feature has not been implemented yet

[ERROR]

Description

The C++ compiler does not support all C++ features yet.

Tips

Try to find a workaround of the problem or ask about the latest version of the compiler.

See also

- C++ section about features that are not implemented yet.

26.1.164 C1402: This C++ feature (<Feature>) is not implemented yet

[ERROR]

Description

The C++ compiler does not support all C++ features yet. Here is the list of features causing the error:

- Compiler generated functions of nested nameless classes
- calling destructors with goto
- explicit operator call
- Create Default Ctor of unnamed class
- Base class member access modification
- local static class obj
- global init with non-const

Tips

Try to avoid to use such a feature, e.g. using global static objects instead local ones.

26.1.165 C1403: Out of memory

[FATAL]

Description

The compiler wanted to allocate memory on the heap, but there was no space left.

Tips

Modify the memory management on your system.

26.1.166 C1404: Return expected

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A function with a return type has no return statement. In C it's a warning, in C++ an error.

Example

```
int f() {}  // warning
```

Tips

Insert a return statement, if you want to return something, otherwise declare the function as returning void type.

26.1.167 C1405: Goto <undeclared label>="> in this function

[ERROR]

Description

A goto label was found, but the specified label did not exist in the same function.

Example

```
void main(void) {


  goto label;


}
```

Tips

Insert a label in the function with the same name as specified in the goto statement.

26.1.168 C1406: Illegal use of identifierList

[ERROR]

Description

A function prototype declaration had formal parameter names, but no types were provided for the parameters.

Example

```
int f(i);  // error
```

Tips

Declare the types for the parameters.

26.1.169 C1407: Illegal function-redefinition

[ERROR]

Description

The function has already a function body, it has already been defined.

Example

```
void main(void) {}


void main(void) {}
```

Tips

Define a function only once.

26.1.170 C1408: Incorrect function-definition

[ERROR]

Description

The function definition is not correct or ill formed.

Tips

Correct the source.

26.1.171 C1409: Illegal combination of parameterlist and identlist

[ERROR]

Description

The parameter declaration for a function does not match with the declaration.

Tips

Correct the source. Maybe a wrong header file has been included.

26.1.172 C1410: Parameter-declaration - identifier-list mismatch

[ERROR]

Description

The parameter declaration for a function does not match with the declaration.

Tips

Correct the source. Maybe a wrong header file has been included.

26.1.173 C1411: Function-definition incompatible to previous declaration

[ERROR]

Description

An old-style function parameter declaration was not compatible to a previous declaration.

Example

```
void f(int i);


void f(i,j) int i; int j; {} // error
```

Tips

Declare the same parameters as in the previous declaration.

26.1.174 C1412: Not a function call, address of a function

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A function call was probably desired, but the expression denotes an address of the function.

Example

```
void f(int i);


void main() {


  f;  // warning


}
```

Tips

Write parenthesis (,) with the arguments after the name of the function to be called.

26.1.175 C1413: Illegal label-redeclaration

[ERROR]

Description

The label was defined more than once.

Example

```
Label:


...
```

```
Label:  // label redefined
```

Tips

Choose another name for the label.

26.1.176 C1414: Casting to pointer of non base class

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A cast was done from a pointer of a class/struct to a pointer to another class/struct, but the second class/struct is not a base of the first one.

Example

```
class A{} a;


class B{};


void main(void) {


  B* b= (B*)&a;


}
```

Tips

Check if your code really does what you want it to.

26.1.177 C1415: Type expected

[ERROR]

Description

The compiler cannot resolve the type specified or no type was specified. This message may also occur if no type is specified for a new operator.

Tips

Correct the source, add a type for the new operator.

26.1.178 C1416: No initializer can be specified for arrays

[ERROR]

Description

An initializer was given for the specified array created with the new operator.

Tips

Initialize the elements of the array after the statement containing the new operator.

26.1.179 C1417: Const/volatile not allowed for type of new operator

[ERROR]

Description

The new operator can only create non const and non volatile objects.

Example

```
void main() {

  int *a;

  typedef const int I;
```

```
    a=new I;  // error


}
```

Tips

Do not use const/volatile qualifiers for the type given to the new operator.

26.1.180 C1418:] expected for array delete operator

[ERROR]

Description

There was no] found after the [of a delete operator.

Example

```
delete [] MyArray;   // ok


delete [3] MyArray;  // error
```

Tips

Add a] after the [.

26.1.181 C1419: Non-constant pointer expected for delete operator

[ERROR]

Description

A pointer to a constant object was illegally deleted using the delete operator.

Example

```
void main() {

  const int *a;

  a=new int;

  delete a; // error

}
```

Tips

The pointer to be deleted has to be non-constant.

26.1.182 C1420: Result of function-call is ignored

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A function call was done without saving the result.

Example

```
int f(void);

void main(void) {

  f(); // ignore result

}
```

Tips

Assign the function call to a variable, if you need the result afterwards. Otherwise cast the result to void. E.g.:

```
int f(void);

void main(void) {

  (void)f(); // explicitly ignore result

}
```

26.1.183 C1421: Undefined class/struct/union

[ERROR]

Description

An undefined class, structure or union was used.

Example

```
void f(void) {

  struct S *p, *p1;

  *p=*p1;

}
```

Tips

Define the class/struct/union.

26.1.184 C1422: No default Ctor available

[ERROR]

Description

No default constructor was available for the specified class/struct. The compiler will supply a default constructor only if user-defined constructors are not provided in the same class/struct and there are default constructors provided in all base/member classes/structs.

Example

```
class A {


  public:


    A(int i); // constructor with non-void parameter


    A();      // default constructor


};
```

Tips

If you provide a constructor that takes a non-void parameter, then you must also provide a default constructor. Otherwise, if you do not provide a default constructor, you must call the constructor with parameters.

Example

```
class A {
```

```
  public:

  A(int i);

};

A a(3); // constructor call with parameters
```

26.1.185 C1423: Constant member must be in initializer list

[ERROR]

Description

There are constant members in the class/struct, that are not initialized with an initializer list in the object constructor.

Example

```
struct A {

  A();

  const int i;

};

A::A() : i(4) {  // initialize i with 4

}
```

Tips

If a const or reference member variable is not given a value when it is initialized, it must be given a value in the object constructor.

26.1.186 C1424: Cannot specify explicit initializer for arrays

[ERROR]

Description

The specified member of the class/struct could not be initialized, because it is an array.

Tips

Initialize the member inside the function body of the constructor.

26.1.187 C1425: No Destructor available to call

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

No destructor is available, but one must be called.

26.1.188 C1426: Explicit Destructor call not allowed here

[ERROR]

Description

Explicit Destructor calls inside member functions without using this are illegal.

Example

```
struct A {
```

```
    void f();

    ~A();

  };

  void A::f() {

    ~A(); // illegal

    this->~A();  // ok

  }
```

Tips

Use the this pointer.

26.1.189 C1427: 'this' allowed in member functions only

[ERROR]

Description

The specified global function did not have a this pointer to access.

Tips

Do not use this in global functions.

26.1.190 C1428: No wide characters supported

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The Compiler does not support wide characters. They are treated as conventional characters.

Example

```
char c= L'a';  // warning
```

Tips

Do not specify the L before the character/string constant.

26.1.191 C1429: Not a destructor id

[ERROR]

Description

Another name than the name of the class was used to declare a destructor.

Tips

Use the same name for the destructor as the class name.

26.1.192 C1430: No destructor in class/struct declaration

[ERROR]

Description

There was no destructor declared in the class/struct.

Example

```
struct A {
```

```
};

A::~A() {}  // error

void main() {

  A.a;

  a.~A();    // legal

}
```

Tips

Declare a destructor in the class/struct.

26.1.193 C1431: Wrong destructor call

[ERROR]

Description

This call to the destructor would require the destructor to be static. But destructors are never static.

Example

```
class A {

public:

  ~A();
```

```
  A();

};

void main() {

  A::~A();  // error

  A::A();   // ok, generating temporary object

}
```

Tips

Do not make calls to static destructors, because there are no static destructors.

26.1.194 C1432: No valid classname specified

[ERROR]

Description

The specified identifier was not a class/structure or union.

Example

```
int i;

void main() {

  i::f();  // error
```

```
}
```

Tips

Use a name of a class/struct/union.

26.1.195 C1433: Explicit Constructor call not allowed here

[ERROR]

Description

An explicit constructor call was done for a specific object.

Example

```
struct A {

  A();

  void f();

};

void A::f() {

  this->A();  // error

  A();        // ok, generating temporary object

}
```

```
void main() {

  A a;

  a.A();     // error

  A();       // ok, generating temporary object

}
```

Tips

Explicit constructor calls are only legal, if no object is specified, that means, a temporary object is generated.

26.1.196 C1434: This C++ feature is not yet implemented

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The C++ compiler does not yet support all C++ features. Here is a list of not yet implemented features causing a warning: Check for NULL ptr for this complex expression Class parameters (for some processors this is already implemented!) Class returns (for some processors this is already implemented!)

Tips

The C++ compiler ignores this C++ feature and behaves like a C-Compiler.

26.1.197 C1435: Return expected

[ERROR]

Description

The specified function was declared as returning a value, but the function definition did not contain a return statement.

Example

```
int foo(void) {}
```

Tips

Write a return statement in this function or declare the function with a void return type.

26.1.198 C1436: delete needs number of elements of array

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A call to the delete[] operator was made without specifying the number of elements of the array, which is necessary for deleting a pointer to a class needing to call a destructor.

Example

```
class A {

  // ...

  ~A();

};

class B {

  // ...
```

```
};

void f(A *ap, B *bp) {

  delete ap;     // ok

  delete[] ap;   // error

  delete[4] ap;  // ok

  delete bp;     // ok

  delete[] bp;   // ok

  delete[4] bp;  // ok

}
```

Tips

Specify the number of elements at calling delete[].

26.1.199 C1437: Member address expected

[ERROR]

Description

A member address is expected to initialize the pointer to member. 0 value can also be provide to set the pointer to member to NULL.

Example

```
class A{

public:

  int a;

  void fct(void){}

};

void main(void){

  int A::*pmi = NULL;

  ...

}
```

Tips

Use a member address

```
class A{

public:

  int a;
```

```
  void fct(void){}

};

void main(void){

  int A::*pmi = &amp;A::a;

  void (A::*pmf)() = &amp;A::fct;

  ...

}
```

Use 0 value to set the pointer to member to NULL

```
class A{

public:

  int a;

  void fct(void){}

};

void main(void){
```

```
    int A::*pmi = 0;

    void (A::*pmf)() = 0;

    ...

}
```

26.1.200 C1438: ... is not a pointer to member ident

[ERROR]

Description

Parsing ident is not a pointer to member as expected.

Example

```
int glob;

class A{

public:

  int a;

  void fct(void){}

};
```

```
void main(void){

  int A::*pmi = &amp;A::a;

  void (A::*pmf)() = &amp;A::fct;

  A aClass;

  ...

  aClass.*glob = 4;

  (aclass.*glob)();

}
```

Tips

Use the pointer to member ident

```
class A{

public:

  int a;

  void fct(void){}
```

```
};

void main(void){

  int A::*pmi = &amp;A::a;

  void (A::*pmf)() = &amp;A::fct;

  A aClass;

  ...

  aClass.*pmi = 4;

  (aclass.*pmf)();

}
```

26.1.201 C1439: Illegal pragma __OPTION_ACTIVE__, <Reason>

[ERROR]

Description

An ill formed __OPTION_ACTIVE__ expression was detected. The reason argument gives a more concrete hint what actually is wrong.

Example

```
#if __OPTION_ACTIVE__("-dABS")
```

```
#endif
```

The __OPTION_ACTIVE__ expression only allows the option to be tested (here -d and not the content of the option here ABS.

Tips

Only use the option. To check if a macro is defined as in the example above, use if defined(ABS). Only options known to the compiler can be tested. This option can be moved to an warning or less.

Seealso

- if __OPTION_ACTIVE__

26.1.202 C1440: This is causing previous message <MsgNumber>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

This message informs about the problem causing the previous message with the number 'MsgNumber'. Because the reason for the previous message may be in another file (e.g. header file), this message helps to find out the problem.

Example

```
void foo(void);


int foo(void) {}


produces following messages:


int foo(void) {}


^
```

```
ERROR C1019: Incompatible type to previous declaration

             (found 'int (*) ()', expected 'void (*) ()')

void foo(void);

^

INFORMATION C1440: This is causing previous message 1019
```

Tips

The problem location is either the one indicated by the previous message or the location indicated by this message.

26.1.203 C1441: Constant expression shall be integral constant expression

[ERROR]

Description

A constant expression which has to be an integral expression is not integral. A non-integral expression is e.g. a floating constant expression.

Example

```
#if 1.   // <&lt; has to be integral!

;

#endif
```

Tips

Use a integral constant expression.

NOTE

If you move this message (to disable/information/warning), the non-integral constant expression is transformed into an integral expression (e.g. `2.3 => 2`).

26.1.204 C1442: Typedef cannot be used for function definition

[ERROR]

Description

A typedef name was used for a function definition.

Example

```
typedef int INTFN();


INTFN f { return (0); }  // <&lt; error
```

Tips

Do not use a typedef name for a function definition.

26.1.205 C1443: Illegal wide character

[ERROR]

Description

There is a illegal wide character after a wide character designator (L).

Example

```
int i = sizeof(L 3.5);
```

Tips

After L there has to be a character constant (e.g. `L'a'`) or a string (e.g. `L"abc"`).

26.1.206 C1444: Initialization of <Variable> is skipped by 'case' label

[ERROR]

Description

Initialization of a local variable is skipped by a 'case' label.

Example

```
void main(void){

  int i;



  switch(i){

    int myVar = 5;

    case 0:          // C1444 init skipped

      //...

      break;
```

```
    case 1:          // C1444 init skipped

      //...

      break;

  }
```

Tips

Declare the local variable in the block where it is used.

```
void main(void){

  int i;



  switch(i){

    case 0:

      //...

      break;

    case 1:
```

```
        int myVar = 5;

        //...

        break;

    }
```

26.1.207 C1445: Initialization of <Variable> is skipped by 'default' label

[ERROR]

Description

Initialization of a local variable is skipped by a 'default' label.

Example

```
  void main(void){

    int i;



    switch(i){

      case 0:

        //...
```

```
      break;

    int myVar = 5;

    default:          // C1445 init skipped

      //...

      break;

  }
```

Tips

Declare the local variable in the block where it is used.

```
void main(void){

  int i;



  switch(i){

    case 0:

      //...
```

```
      break;

    default:

      int myVar = 5;

      //...

      break;

  }
```

26.1.208 C1800: Implicit parameter-declaration (missing prototype) for '<FuncName>'

[ERROR]

Description

A function was called without its prototype being declared before.

Example

```
void f(void) {

  g();

}
```

```
This message is only used for C++ or for C if option \c
-Wpd, but not Option \c -Ansi is given
```

Tips

Prototype the function before calling it. Use void to define a function with no parameters.

```
void f(); // better: 'void f(void)'


void g(void);
```

The C the declaration f does not define anything about the parameters of f. The first time f is used, the parameters get defined implicitly. The function g is defined to have no parameters.

The C the declaration f does not define anything about the parameters of f. The first time f is used, the parameters get defined implicitly. The function g is defined to have no parameters.

26.1.209 C1801: Implicit parameter-declaration for '<FuncName>'

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A function was called without its prototype being declared before.

Example

```
void f(void) {


  g();


}
```

Tips

Prototype the function before calling it. Make sure that there is a prototype/declaration for the function. E.g. for above example:

```
void g(void); // having correct prototype for 'g'


void f(void) {


  g();


}
```

26.1.210 C1802: Must be static member

[ERROR]

Description

A non-static member has been accessed inside a static member function.

Example

```
struct A {


  static void f();


  int i;


};


void A::f() {
```

```
    i=3;  // error


}
```

Tips

Remove the static specifier from the member function, or declare the member to be accessed as static.

26.1.211 C1803: Illegal use of address of function compiled under the pragma REG_PROTOTYPE

[ERROR]

Description

A function compiler with the pragma REG_PROTOTYPE was used in a unsafe way.

26.1.212 C1804: Ident expected

[ERROR]

Description

An identifier was expected.

Example

```
int ;
```

26.1.213 C1805: Non standard conversion used

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

In ANSI-C it is normally not allowed to cast an object pointer to a function pointer or a function pointer to an object pointer. Note that the layout of a function pointer may not be the same than the layout of a object pointer.

Example

```
typedef unsigned char (*CmdPtrType)


        (unsigned char, unsigned char);


typedef struct STR {int a;} baseSTR;


baseSTR strDescriptor;


CmdPtrType myPtr;


// next line does not strictly correspond to ANSI C


//   but we make sure that the correct cast is being used


void foo(void) {


  myPtr=(CmdPtrType)(void*)&amp;strDescriptor; //
message C1805


}


*/


/*! \page pageC1806 C1806: Illegal cast-operation
```

```
[ERROR]
```

Description

There is no conversion for doing the desired cast.

Tips

The cast operator must specify a type, that can be converted to the type, which the expression containing the cast operator would be converted to.

26.1.214 C1806: Illegal cast-operation

[ERROR]

Description

There is no conversion for doing the desired cast.

Tips

The cast \c operator must specify a type, that can be converted to the type, which the expression containing the cast \c operator would be converted to.

26.1.215 C1807: No conversion to non-base class

[ERROR]

Description

There is no conversion from a class to another class that is not a base class of the first class.

Example

```
struct A {


  int i;
```

```
};

struct B : A {

  int j;

};

void main() {

  A a;

  B b;

  b=a;  // error:  B is not a base class of A

}
```

Tips

Remove this statement, or modify the class hierarchy.

26.1.216 C1808: Too many nested switch-statements

[ERROR]

Description

The number of nested switches was too high.

Example

```
switch(i0) {

  switch(i1) {

    switch(i2) {

    ...
```

Tips

Use "if-else if" statements instead or reduce nesting level.

See also

- Limitations

26.1.217 C1809: Integer value for switch-expression expected

[ERROR]

Description

The specified switch expression evaluated to an illegal type.

Example

```
float f;

void main(void) {

  switch(f) {

  case 1:
```

```
      f=2.1f;

      break;

    case 2:

      f=2.1f;

      break;

    }

  }
```

Tips

A switch expression must evaluate to an integral type, or a class type that has an unambiguous conversion to an integral type.

26.1.218 C1810: Label outside of switch-statement

[ERROR]

Description

The keyword case was used outside a switch.

Tips

The keyword case can appear only within a switch statement.

26.1.219 C1811: Default-label twice defined

[ERROR]

Description

A switch statement must have no or one default label. Two default labels are indicated by this error.

Example

```
switch (i) {

  default: break;

  case 1: f(1); break;

  case 2: f(2); break;

  default: f(0); break;

}
```

Tips

Define the default label only once per switch-statement.

26.1.220 C1812: Case-label-value already present

[ERROR]

Description

A case label value is already present.

Tips

Define a case-label-value only once for each value. Use different values for different labels.

26.1.221 C1813: Division by zero

[ERROR]

Description

A constant expression was evaluated and found to have a zero denominator.

Example

```
int i = 1/0;
```

Tips

mod or divide should never be by zero. Note that this error can be changed to a warning or less. This way code like the following can be compiled:

```
int i = (sizeof(int) == sizeof(long)) ? 0 : sizeof(long)
/ (sizeof(long)-sizeof(int))
```

26.1.222 C1814: Arithmetic or pointer-expression expected

[ERROR]

Description

The expression had an illegal type!.

Tips

Expressions after a ! operator and expressions in conditions must be of arithmetic or pointer type.

26.1.223 C1815: <Name> not declared (or typename)

[ERROR]

Description

The specified identifier was not declared.

Example

```
void main(void) {


  i=2;


}
```

Tips

A variable's type must be specified in a declaration before it can be used. The parameters that a function uses must be specified in a declaration before the function can be used. This error can be caused, if an include file containing the required declaration was omitted.

26.1.224 C1816: Unknown struct- or union-member

[ERROR]

Description

A nonmember of a structure or union was incorrectly used.

Example

```
struct A {


  int i;


} a;
```

```
void main(void) {


  a.I=2;


}
```

Tips

On the right side of the ì->ì or . operator, there must be a member of the structure/union specified on the left side. C is case sensitive.

26.1.225 C1817: Parameter cannot be converted to non-constant reference

[ERROR]

Description

A constant argument was specified for calling a function with a reference parameter to a non-constant.

Example

```
void f(const int &amp;);  // ok


void f(int &amp;); // causes error, when calling


              // with constant argument


void main() {


  f(3);    // error for second function declaration
```

```
}
```

Tips

The parameter must be a reference to a constant, or pass a non-constant variable as argument.

26.1.226 C1819: Constructor call with wrong number of arguments

[ERROR]

Description

The number of arguments for the constructor call at a class object initialization was wrong.

Example

```
struct A {

  A();

};

void main() {

  A a(3);   // error

}
```

Tips

Specify the correct number of arguments for calling the constructor. Try to disable the option -Cn=Ctr, so the compiler generates a copy constructor, which may be required in your code.

26.1.227 C1820: Destructor call must have 'void' formal parameter list

[ERROR]

Description

A destructor call was specified with arguments.

Example

```
struct A {

  ~A();

};

void main() {

  A a;

  a.~A(3);   // error

}
```

Tips

Destructor calls have no arguments!

26.1.228 C1821: Wrong number of arguments

[ERROR]

Description

A function call was specified with the wrong number of formal parameters.

Example

```
struct A {

 void f();

};

void main() {

  A a;

  a.f(3);    // error

}
```

Tips

Specify the correct number of arguments.

26.1.229 C1822: Type mismatch

[ERROR]

Description

There is no conversion between the two specified types.

Example

```
void main() {

  int *p;

  int j;

  p=j;     // error

}
```

Tips

Use types that can be converted.

26.1.230 C1823: Undefining an implicit parameter-declaration

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A implicit parameter declaration was removed because of a assignment.

Example

```
void (*f)();

void g(long );
```

```
void main(void) {

  f(1);

  f=g;

  f(2);

}
```

Tips

Avoid implicit parameter declarations whenever possible.

26.1.231 C1824: Indirection to different types

[ERROR]

Description

There are two pointers in the statement pointing to non-equal types.

Example

```
void main() {

  int *i;

  const int *ci;

  char *c;
```

```
    i=ci;   // C: warning, C++ error, C++ -ec warning


    i=c;    // C: warning, C++: error


  }
```

Tips

Both pointers must point to equal types. If the types only differ in the qualifiers (const, volatile) try to compile with the option -ec.

26.1.232 C1825: Indirection to different types

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

There are two pointers in the statement pointing to non-equal types.

Example

```
  void main() {


    int *i;


    char *c;


    i=c;  // C: warning, C++: error


  }
```

Tips

Both pointers should point to equal types.

26.1.233 C1826: Integer-expression expected

[ERROR]

Description

The expression was not of integral type.

Example

```
void main() {

  int *p;

  p<&lt;3;// error

}
```

Tips

The expression must be an integral type.

26.1.234 C1827: Arithmetic types expected

[ERROR]

Description

After certain operators as * or /, arithmetic types must follow.

Example

```
void main() {
```

```
    int * p;


    p*3;   // error


  }
```

Tips

* and / must have operands with arithmetic types.

26.1.235 C1828: Illegal pointer-subtraction

[ERROR]

Description

A pointer was subtracted from an integral type.

Example

```
  void main() {


    int *p;


    int i;


    i-p;    // error


  }
```

Tips

Insert a cast operator from the pointer to the integral type.

26.1.236 C1829: + - incompatible Types

[ERROR]

Description

For + and - only compatible types on both sides can be used. In C++ own implementation can be used with overloading.

Example

```
struct A {

  int i;

};

void main() {

  int i;

  A a;

  i=i+a;   // error

}
```

Tips

Use compatible types on the left and on the right side of the +/- operator. Or use the operator-overloading and define an own + or - operator!

26.1.237 C1830: Modifiable lvalue expected

[ERROR]

Description

An attempt was made to modify an item declared with const type.

Example

```
const i;


void main(void) {


  i=2;


}
```

Tips

Do not modify this item, or declare the item without the const qualifier.

26.1.238 C1831: Wrong type or not an lvalue

[ERROR]

Description

An unary operator has an operand of wrong or/and constant type.

Tips

The operand of the unary operator must be a non-const integral type or a non-const pointer to a non-void type. Or use operator overloading!.

26.1.239 C1832: Const object cannot get incremented

[ERROR]

Description

Constant objects can not be changed.

Example

```
int* const pi;


void main(void) {


  *pi++;


}
```

Tips

Either do not declare the object as constant or use a different constant for the new value. In the case above, use parenthesis to increment the value pi points to and to not increment pi itself.

```
int* const pi;


void main(void) {


  (*pi)++;


}
```

26.1.240 C1833: Cannot take address of this object

[ERROR]

Description

An attempt to take the address of an object without an address was made.

Example

```
void main() {

  register i;

  int *p=&amp;i;   // error

}
```

Tips

Specify the object you want to dereference in a manner that it has an address.

26.1.241 C1834: Indirection applied to non-pointer

[ERROR]

Description

The indirection operator (*) was applied to a non-pointer value.

Example

```
void main(void) {

  int i;
```

```
  *i=2;


}
```

Tips

Apply the indirection operator only on pointer values.

26.1.242 C1835: Arithmetic operand expected

[ERROR]

Description

The unary (-) operator was used with an illegal operand type.

Example

```
const char* p= -"abc";
```

Tips

There must be an arithmetic operand for the unary (-) operator.

26.1.243 C1836: Integer-operand expected

[ERROR]

Description

The unary (~) operator was used with an illegal operand type.

Example

```
float f= ~1.45;
```

Tips

There must be an operand of integer type for the unary (~) operator.

26.1.244 C1837: Arithmetic type or pointer expected

[ERROR]

Description

The conditional expression evaluated to an illegal type.

Tips

Conditional expressions must be of arithmetic type or pointer.

26.1.245 C1838: Unknown object-size: sizeof (incomplete type)

[ERROR]

Description

The type of the expression in the sizeof operand is incomplete.

Example

```
int i = sizeof(struct A);
```

Tips

The type of the expression in the sizeof operand must be defined complete.

26.1.246 C1839: Variable of type struct or union expected

[ERROR]

Description

A variable of structure or union type was expected.

26.1.247 C1840: Pointer to struct or union expected

[ERROR]

Description

A pointer to a structure or union was expected.

26.1.248 C1842: [incompatible types

[incompatible types [ERROR]

Description

Binary operator [: There was no global operator defined, which takes the type used.

Example

```
struct A {

  int j;

  int operator [] (A a);

};

void main() {

  A a;

  int i;
```

```
    int b[3];


    i=a[a];  // ok


    i=b[a];  // error


}
```

Tips

Use a type compatible to ì[ì. If there is no global operator for [, take an integer type.

26.1.249 C1843: Switch-expression: integer required

[ERROR]

Description

Another type than an integer type was used in the switch expression.

Tips

Use an integer type in the switch expression.

26.1.250 C1844: Call-operator applied to non-function

[ERROR]

Description

A call was made to a function through an expression that did not evaluate to a function pointer.

Example

```
int i;
```

```
void main(void) {

  i();

}
```

Tips

The error is probably caused by attempting to call a non-function. In C++ classes can overload the call operator, but basic types as pointers cannot.

26.1.251 C1845: Constant integer-value expected

[ERROR]

Description

The case expression was not an integral constant.

Example

```
int i;

void main(void) {

  switch (i) {

    case i+1:

      i=1;

  }
```

```
}
```

Tips

Case expressions must be integral constants.

26.1.252 C1846: Continue outside of iteration-statement

[ERROR]

Description

A continue was made outside of an iteration-statement.

Tips

The continue must be done inside an iteration-statement.

26.1.253 C1847: Break outside of switch or iteration-statement

[ERROR]

Description

A break was made outside of an iteration-statement.

Example

```
int i;


void f(void) {


  int res;


  for (i=0; i < 10; i++ )
```

```
    res=f(-1);

    if (res == -1)

      break;

    printf("%d\n", res);

}
```

Tips

The break must be done inside an iteration-statement. Check for the correct number of open braces.

26.1.254 C1848: Return <expression> expected

[ERROR]

Description

A return was made without an expression to be returned in a function with a non-void return type.

Tips

The return statement must return an expression of the return-type of the function.

26.1.255 C1849: Result returned in void-result-function

[ERROR]

Description

A return was made with an expression, though the function has void return type.

Example

```
void f(void) {


  return 1;


}
```

Tips

Do not return an expression in a function with void return type. Just write return, or write nothing.

26.1.256 C1850: Incompatible pointer operands

[ERROR]

Description

Pointer operands were incompatible.

Tips

Either change the source or explicitly cast the pointers.

26.1.257 C1851: Incompatible types

[ERROR]

Description

Two operands of a binary operator did not have compatible types (there was no conversion, or the overloaded version of the operand does not take the same types as the formal parameters).

Tips

Both operands of the binary operator must have compatible types.

26.1.258 C1852: Illegal sizeof operand

[ERROR]

Description

The sizeof operand was a bitfield.

Tips

Do not use bitfields as sizeof operands.

26.1.259 C1853: Unary minus operator applied to unsigned type

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

The unary minus operator was applied to an unsigned type.

Example

```
void main(void) {

  unsigned char c;

  unsigned char d= -c;

}
```

Tips

An unsigned type never can become a negative value. So using the unary minus operator may cause an unwanted behavior! Note that ANSI C treats -1 as negated value of 1. Therefore 2147483648 is an unsigned int, if int is 32 bits large or an unsigned long if not. The negation is a unary function as any other, so the result type is the argument type

propagated to int, if smaller. Note that the value -2147483648 is the negation of 2147483648 and therefore also of a unsigned type, even if the signed representation contains this value.

26.1.260 C1854: Returning address of local variable

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

An address of a local variable is returned.

Example

```
int &amp;f(void) {

  int i;

  return i;   // warning

}
```

Tips

Either change the return type of the function to the type of the local variable returned, or declare the variable to be returned as global (returning the reference of this global variable)!

26.1.261 C1855: Recursive function call

[DISABLE, INFORMATION, **WARNING** , ERROR]

Description

A recursive function call was detected. There is a danger of endless recursion, which leads to a stack overflow.

Example

```
void f(void) {


  f();  // warning; this code leads to an endless
recursion


}


void g(int i) {


  if(i>0) {


    g(--i);  // warning; this code has no endless
recursion


  }


}
```

Tips

Be sure there is no endless recursion. This would lead to a stack overflow.

26.1.262 C1856: Return <expression> expected

[DISABLE, ERROR, **WARNING** , INFORMATION]

Description

A return statement without expression is executed while the function expects a return value. In ANSI-C, this is correct but not clean. In such a case, the program runs back to the caller. If the caller uses the value of the function call then the behavior is undefined.

Example

```
int foo(void){

  return;

}

void main(void){

int a;

  ...

  a = foo();

  ...          // behavior undefined

}
```

Tips

```
#define ERROR_CASE_VALUE 0

int foo(void){

  return ERROR_CASE_VALUE;   // return something...
```

```
}

void main(void){

int a;

  ...

  a = foo();

  if (a==ERROR_CASE_VALUE){  // ... and treat this case

    ...

  } else {

    ...

  }

  ...

}
```

26.1.263 C1857: Access out of range

[DISABLE, ERROR, **WARNING** , INFORMATION]

Description

The compiler has detected that there is an array access outside of the array bounds. This is legal in ANSI-C/C++, but normally it is a programming error. Check carefully such accesses that are out of range. This warning does not check the access, but also taking the address out of an array. However, it is legal in C to take the address of one element outside the array range, so there is no warning for this. Because array accesses are treated internally like address-access operations, there is no message for accessing on element outside of the array bounds.

Example

```
char buf[3], *p;


p = &amp;buf[3]; // no message!


buf[4] = 0;  // message
```

26.1.264 C1858: Partial implicit parameter-declaration

[WARNING]

Description

A function was called without its prototype being totally declared before.

Example

```
void foo(); // not complete prototype, arguments not
known


void main(void) {


  foo();
```

```
}
```

Tips

Prototype all arguments of the function before calling it.

26.1.265 C1859: Indirection operator is illegal on Pointer To Member operands

[ERROR]

Description

It is illegal to apply indirection '*' operator to Pointer To Member operands.

Example

```
class A {

public:

  void f(void) {}

};

typedef void (A::*ptrMbrFctType)(void);

void fct0(void){

  ptrMbrFctType pmf;

  *pmf=A::f; // ERROR
```

```
}

void fct1(void){

  void (* A::*pmf)(void)=A::f; // ERROR

}
```

Tips

Remove the indirection operator.

```
class A {

public:

  void f(void) {}

};

typedef void (A::*ptrMbrFctType)(void);

void fct0(void){

  ptrMbrFctType pmf;

  pmf=&amp;A::f;
```

```
}

void fct1(void){

  void (A::*pmf)(void)=&amp;A::f;

}
```

26.1.266 C1860: Pointer conversion: possible loss of data

[WARNING]

Description

Whenever there is a pointer conversion which may produce loss of data, this message is produces. Loss of data can happen if a far (e.g. 3 byte pointer) is assigned to a pointer of smaller type (e.g. a near 1 byte pointer).

Example

```
char *near nP;

char *far fP;

void foo(void) {

  nP = fP; // warning here

}
```

Tips

Check if this loss of data is intended.

26.1.267 C1861: Illegal use of type 'void'

[WARNING]

Description

The compiler has detected an illegal usage of the void type. The compiler accepts this because of historical reasons. Some other vendor compilers may not accept this at all, so this may cause portability problems.

Example

```
int foo(void buf[256]) { // warning here


  ...


}
```

Tips

Correct your code. E.g. replace in the above example the argument with 'void *buf'.

26.1.268 C2000: No constructor available

[ERROR]

Description

A constructor must be called, but none is available.

Tips

Define a constructor. No compiler defined default constructor is defined in some situations, for example when the class has constant members.

26.1.269 C2001: Illegal type assigned to reference.

[ERROR]

Description

There is no conversion from type assigned to the type of the reference.

Example

```
int *i;


int &amp;r=i;   // error
```

Tips

The type of the reference must be equal to the type assigned.

26.1.270 C2004: Non-volatile reference initialization with volatile illegal

[ERROR]

Description

The reference type is not volatile, the assigned type is.

Example

```
volatile i;


const int &amp;r=i;  // illegal
```

Tips

Either both are volatile or both are not volatile.

26.1.271 C2005: Non-constant reference initialization with constant illegal

[ERROR]

Description

The reference type is not constant, the assigned type is.

Example

```
void main(void) {


  const int i=1;


  int  &amp;p=i;


}
```

Tips

Either both are const or both are not const.

26.1.272 C2006: (un)signed char reference must be const for init with char

[ERROR]

Description

The initializer for a reference to a signed or unsigned char must be const for initialization with a plain char.

Example

```
char i;


signed char &amp;r=i;    // error
```

Tips

Either declare the reference type as const, or the type of the initializer must not be plain.

26.1.273 C2007: Cannot create temporary for reference in class/ struct

[ERROR]

Description

A member initializer for a reference was constant, though the member was non-constant

Example

```
struct A {


  int &amp;i;


  A();


};


A::A() : i(3) {  // error


}
```

Tips

Initialize the reference with a non-constant variable.

26.1.274 C2008: Too many arguments for member initialization

[ERROR]

Description

The member-initializer contains too many arguments.

Example

```
struct A {

  const int i;

  A();

};

A::A() : i(3,5) {  // error

}
```

Tips

Supply the correct number of arguments in the initializer list of a constructor.

26.1.275 C2009: No call target found!

[ERROR]

Description

The ambiguity resolution mechanism did not find a function in the scope, where it expected one.

Tips

Check, if the function called is declared in the correct scope.

26.1.276 C2010: <Name> is ambiguous

[ERROR]

Description

The ambiguity resolution mechanism found an ambiguity. That means, more than one object could be taken for the identifier Name. So the compiler does not know which one is desired.

Example

```
struct A {

  int i;

};

struct B : A {

};

struct C : A {

};
```

```
struct D : B, C {


};


void main() {


  D d;


  d.i=3;   // error, could take i from B::A or C::A


  d.B::i=4; // ok


  d.C::i=5; // ok


}
```

Tips

Specify a path, how to get to the desired object. Or use virtual base classes in multiple inheritance. The compiler can handle a most 10'000 different numbers for a compilation unit. Internally for each number a descriptor exists. If an internal number descriptor already exists for a given number value with a given type, the existing one is used. But if e.g. more than 10'000 different numbers are used, this message will appear.

26.1.277 C2011: <Name> can not be accessed

[ERROR]

Description

There is no access to the object specified by the identifier.

Example

```
struct A {

private:

  int i;

protected:

  int j;

public:

  int k;

  void g();

};

struct B : public A {

  void h();

};

void A::g() {
```

```
  this->i=3;  // ok

  this->j=4;  // ok

  this->k=5;  // ok

}

void B::h() {

  this->i=3;  // error

  this->j=4;  // ok

  this->k=5;  // ok

}

void f() {

  A a;

  a.i=3;  // error

  a.j=4;  // error
```

```
    a.k=5;  // ok


}
```

Tips

Change the access specifiers in the class declaration, if you really need access to the object. Or use the friend-mechanism.

See also

- Limitations

26.1.278 C2012: Only exact match allowed yet or ambiguous!

[ERROR]

Description

An overloaded function was called with non-exact matching arguments.

Tips

Supply exact matching parameters.

See also

- Limitations

26.1.279 C2013: No access to special member of base class

[WARNING]

Description

The special member (Constructor, destructor or assignment operator) could not be accessed.

Example

```
struct A {

  private:

    A();

};

struct B : A {

};

void main() {

  B b;   // error

}
```

Tips

Change the access specifier for the special member.

See also

- Limitations

26.1.280 C2014: No access to special member of member class

[WARNING]

Description

The special member (Constructor, destructor or assignment operator) could not be accessed.

Example

```
struct A {

  private:

    A();

};

struct B  {

  A a;

};

void main() {

  B b;  // error

}
```

Tips

Change the access specifier for the special member.

See also

- Limitations

26.1.281 C2015: Template is used with the wrong number of arguments

[ERROR]

Description

A template was instantiated with the wrong number of template arguments.

Example

```
template<class S> struct A {

  S s;

};

A<int, int> a;  // error
```

Tips

The instantiation of a template type must have the same number of parameters as the template specification.

26.1.282 C2016: Wrong type of template argument

[ERROR]

Description

A template was instantiated with the wrong type template arguments.

Example

```
template<class S> struct A {
```

```
  S s;


};


A<4> a;  // error
```

Tips

The instantiation of a template type must have the same argument type as the ones template specification.

26.1.283 C2017: Use of incomplete template class

[ERROR]

Description

A template was instantiated from an incomplete class.

Example

```
template<class S> struct A;


A<int, int> a;  // error
```

Tips

The template to be instantiated must be of a defined class.

26.1.284 C2018: Generate class/struct from template

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A class was instantiated from a template.

Example

```
template<class S> struct A {

  S s;

};

A<int> a;  // information
```

26.1.285 C2019: Generate function from template

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A function was instantiated from a template.

Example

```
template<class S> void f(S s) {

  // ...

};

void main(void) {

  int i;
```

```
  char ch;

  f(4);   // generate

  f(i);   // not generating, already generated

  f(ch);  // generate

}
```

Tips

The fewer functions are instantiated from a template, the less code is produced. So try to use already generated template functions instead of letting the compiler generate new ones.

26.1.286 C2020: Template parameter not used in function parameter list

[ERROR]

Description

A template parameter didn't occur in the parameter list of the template function.

Example

```
template<class S> void f(int i) {  // error

  // ...

};
```

Tips

The parameter list of the template function must contain the template parameters.

26.1.287 C2021: Generate NULL-check for class pointer

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

Operations with pointers to related classes always need separate NULL-checks before adding offsets from base classes to inherited classes.

Example

```
class A {

};

class B : public A{

};

void main() {

  A *ap;

  B *bp;

  ap=bp;  // warning
```

```
}
```

Tips

Try to avoid operations with pointers to different, but related classes.

26.1.288 C2022: Pure virtual can be called only using explicit scope resolution

[ERROR]

Description

Pure virtual functions can be called from a constructor of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is an error.

Example

```
class A{

public:

  virtual void f(void) = 0;

  A(){

    f();

  }

};
```

Tips

A pure virtual can be defined. It can be called using explicit qualification only.

```
class A{

public:

  virtual void f(void) = 0;

  A(){

    A::f();

  }

};

void A::f(void){ // defined somewhere

  //...

}
```

26.1.289 C2023: Missing default parameter

[ERROR]

Description

A subsequent parameter of a default parameter is not a default parameter.

Example

```
void f(int i=0, int j);   // error


void f(int i=0, int j=0); // ok
```

Tips

All subsequent parameters of a default parameter must be default, too.

See also

- Overloading.

26.1.290 C2024: Overloaded operators cannot have default arguments

[ERROR]

Description

An overloaded operator was specified with default arguments.

Example

```
struct A{


// ...


};


operator + (A a, int i=0);  // error
```

Tips

Overloaded operators cannot have default arguments. Declare several versions of the operator with different numbers of arguments.

See also

- Overloading.

26.1.291 C2025: Default argument expression can only contain static or global objects or constants

[ERROR]

Description

A local object or non-static class member was used in the expression for a default argument.

Example

```
struct A {

  int t;

  void f(int i=t);  // error

};

void g(int i, int j=i); // error
```

Tips

Only use static or global objects or constants in expressions for default arguments.

26.1.292 C2200: Reference object type must be const

[ERROR]

Description

If a reference is initialized by a constant, the reference has to be constant as well

Example

```
int &amp;ref = 4; // err
```

Tips

Declare the reference as constant.

See also

- Limitations

26.1.293 C2201: Initializers have too many dimensions

[ERROR]

Description

An initialization of an array of class objects with constructor call arguments was having more opening braces than dimensions of the array.

Example

```
struct A {

  A(int);

};

void main() {

  A a[3]={{3,4},4}; // errors
```

```
  A a[3]={3,4,4};   // ok


}
```

Tips

Provide the same number of opening braces in an initialization of an array of class objects.

See also

- Limitations

26.1.294 C2202: Too many initializers for global Ctor arguments

[ERROR]

Description

An initialization of a global array of class objects with constructor call arguments was having more arguments than elements in the array.

Example

```
struct A {


  A(int);


};


A a[3]={3,4,5,6};  // errors


A a[3]={3,4,5};    // ok
```

Tips

Provide the same number of arguments than number of elements in the global array of class objects. If you want to make calls to constructors with more than one argument, use explicit calls of constructors.

See also

- Limitations

26.1.295 C2203: Too many initializers for Ctor arguments

[ERROR]

Description

An initialization of an array of class objects with constructor call arguments was having more arguments than elements in the array.

Example

```
struct A {

  A(int);

};

void main() {

  A a[3]={3,4,5,6};  // errors

  A a[3]={3,4,5};  // ok

}
```

Tips

Provide the same number of arguments than number of elements in the array of class objects. If you want to make calls to constructors with more than one argument, use explicit calls of constructors.

```
struct A {

  A(int);

  A();

  A(int,int);

};

void main() {

  A a[3]={A(3,4),5,A()};

    // first element calls A::A(int,int)

    // second element calls A::A(int)

    // third element calls A::A()

}
```

See also

- Limitations

26.1.296 C2204: Illegal reinitialization

[ERROR]

Description

The variable was initialized more than once.

Example

```
extern int i=2;


int i=3;   // error
```

Tips

A variable must be initialized at most once.

See also

- Limitations

26.1.297 C2205: Incomplete struct/union, object can not be initialized

[ERROR]

Description

Incomplete struct/union, object can not be initialized

Example

```
struct A;


extern A a={3};  // error
```

Tips

Do not initialize incomplete struct/union. Declare first the struct/union, then initialize it.

See also

- Limitations

26.1.298 C2206: Illegal initialization of aggregate type

[ERROR]

Description

An aggregate type (array or structure/class) was initialized the wrong way.

Tips

Use the braces correctly.

See also

- Limitations

26.1.299 C2207: Initializer must be constant

[ERROR]

Description

A global variable was initialized with a non-constant.

Example

```
int i;


int j=i;  // error


or


void function(void){
```

```
    int local;

    static int *ptr = &amp;local;

    // error: address of local can be different

    // in each function call.

    // At second call of this function *ptr is not the same!

}
```

Tips

In C, global variables can only be initialized by constants. If you need non-constant initialization values for your global variables, create an InitModule() function in your compilation unit, where you can assign any expression to your globals. This function should be called at the beginning of the execution of your program. If you compile your code with C++, this error won't occur anymore! In C, initialization of a static variables is done only once. Initializer is not required to be constant by ANSI-C, but this behavior will avoid troubles hard to debug.

See also

- Limitations

26.1.300 C2209: Illegal reference initialization

[ERROR]

Description

A reference was initialized with a braced {, } initializer.

Example

```
struct A {

  int i;

};

A &amp;ref = {4};  // error

A a = {4};     // ok

A &amp;ref2 = a;   // ok
```

Tips

References must be initialized with non-braced expressions.

Seealso

- Limitations

26.1.301 C2210: Illegal initialization of non-aggregate type

[ERROR]

Description

A class without a constructor and with non-public members was initialized.

Tips

Classes with non-public members can only be initialized by constructors.

See also

- Limitations

26.1.302 C2211: Initialization of a function

[ERROR]

Description

A function was initialized.

Example

```
void f()=3;
```

Tips

Functions cannot be initialized. But function pointers can.

See also

- Limitations

26.1.303 C2212: Initializer may be not constant

[ERROR]

Description

A global variable was initialized with a non-constant.

Example

```
int i;


int j=i;  // error


or


void function(void){
```

```
    int local;


    static int *ptr = &amp;local;


    // error: address of local can be different


    // in each function call.


    // At second call of this function *ptr is not the same!


}
```

Tips

In C, global variables can only be initialized by constants. If you need non-constant initialization values for your global variables, create an InitModule() function in your compilation unit, where you can assign any expression to your globals. This function should be called at the beginning of the execution of your program. If you compile your code with C++, this error won't occur anymore! In C, initialization of a static variables is done only once. Initializer is not required to be constant by ANSI-C, but this behavior will avoid troubles hard to debug. You can disable this error if your initialization turns out to be constant

See also

- Limitations

26.1.304 C2401: Pragma <ident> expected

[WARNING]

Description

A single pragma was found.

Example

```
#pragma
```

Tips

Probably this is a bug. Correct it.

26.1.305 C2402: Variable <Ident> <State>

[**DISABLE** , INFORMATION, WARNING, ERROR]

Description

A variable was allocated differently because of a pragma INTO_ROM or a pragma FAR.

Example

```
#pragma INTO_ROM


const int i;
```

Tips

Be careful with the pragmas INTO_ROM and pragma FAR. They are only valid for one single variable. In the following code the pragma INTO_ROM puts var_rom into the rom, but var_ram not.

```
#pragma INTO_ROM


const int var_rom, var_ram;
```

Note that pragma INTO_ROM is only for the HIWARE Object file format.

26.1.306 C2450: Expected:

[]

C2450: Expected:

[ERROR]

Description

An unexpected token was found.

Example

```
void f(void);


void main(void) {


  int i=f(void); // error: "void" is an unexpected
keyword!


}
```

Tips

Use a token listed in the error message. Check if you are using the right compiler language option. E.g. you may compile a file with C++ keywords, but are not compiling the file with C++ option set. Too many nested scopes

26.1.307 C2550: Too many nested scopes

[FATAL]

Description

Too many scopes are open at the same time. For the actual limitation number, please see chapter Limitations

Example

```
void main(void) {
```

```
    {

      {

        {


....
```

Tips

Use less scopes.

See also

- Limitations

26.1.308 C2700: Too many numbers

[FATAL]

Description

Too many different numbers were used in one compilation unit. For the actual limitation number, please see chapter Limitations

Example

```
int main(void) {


  return 1+2+3+4+5+6+.....


}
```

Tips

Split up very large compilation units.

See also

- Limitations

26.1.309 C2701: Illegal floating-point number

[WARNING]

Description

An illegal floating point number has been specified or the exponent specified is to large for floating number.

Example

```
float f = 3.e345689;
```

Tips

Correct the floating point number.

See also

- Number Formats
- header file "float.h"

26.1.310 C2702: Number too large for float

[ERROR]

Description

A float number larger than the maximum value for a float has been specified.

Example

```
float f = 3.402823466E+300F;
```

Tips

Correct the number.

See also

- Number Formats
- header file "float.h"

26.1.311 C2703: Illegal character in float number

[ERROR]

Description

The floating number contains an illegal character. Legal characters in floating numbers are the postfixes 'f' and 'F' (for float) or 'l' and 'L' (for long double). Valid characters for exponential numbers are 'e' and 'E'.

Example

```
float f = 3.x4;
```

Tips

Correct the number.

See also

- Number Formats

26.1.312 C2704: Illegal number

[ERROR]

Description

An illegal immediate number has been specified.

Example

```
int i = 4x; // error
```

```
float f= 12345678901234567890;//error too large for a
long!
```

```
float f= 12345678901234567890.;//OK, doubles can be as
large
```

Tips

Correct the number. For floating point numbers, specify a dot.

See also

- Number Formats

26.1.313 C2705: Possible loss of data

[WARNING]

Description

The compiler generates this message if a constant is used which exceeds the value for a type. Another reason for this message is if a object (e.g. long) is assigned to an object with smaller size (e.g. char). Another example is to pass an actual argument too large for a given formal argument, e.g. passing a 32bit value to a function which expects a 8bit value.

Example

```
signed char ch = 128; // char holds only -128..127
```

```
char c;
```

```
long L;
```

```
void foo(short);
```

```
void main(void) {

  c = L; // possible lost of data

  foo(L); // possible lost of data

}
```

Tips

Usually this is a programming error.

See also

- Header file "limits.h"

26.1.314 C2706: Octal Number

[WARNING]

Description

An octal number was parsed.

Example

```
int f(void) {

  return 0100; // warning

}
```

Tips

If you want to have a decimal number, don't write a '0' at the beginning.

26.1.315 C2707: Number too large

[ERROR]

Description

While reading a numerical constant, the compiler has detected the number is too large for a data type.

Example

```
x: REAL;


x := 300e51234;
```

Tips

Reduce the numerical constant value, or choose another data type.

26.1.316 C2708: Illegal digit

[ERROR]

Description

While reading a numerical constant, an illegal digit has been found.

Example

```
x: REAL;


x := 123e4a;
```

Tips

Check your numerical constant for correctness.

26.1.317 C2709: Illegal floating-point exponent ('-', '+' or digit expected)

[ERROR]

Description

While reading a numerical constant, an illegal exponent has been found.

Example

```
x = 123e;
```

Tips

Check your numerical constant for correctness. After the exponent, there has to be an optional '+' or '-' sign followed by a sequence of digits.

26.1.318 C2800: Illegal operator

[ERROR]

Description

An illegal operator has been found. This could be caused by an illegal usage of saturation operators, e.g. the using saturation operators without enabling them with a compiler switch if available. Note that not all compiler backends support saturation operators.

Example

```
i = j +? 3; // illegal usage of saturation \c operator
```

Tips

Enable the usage of Saturation operators if available.

See also

- Saturation Operator
- Compiler Backend Chapter

26.1.319 C2801: <Symbol> missing"

[ERROR]

Description

There is a missing symbol for the Compiler to complete a parsing rule. Normally this is just a closing parenthesis or a missing semicolon.

Example

```
void main(void) {


// '}' missing
```

other example

```
void f() {


  int i


}
```

Tips

Usually this is a programming error. Correct your source code.

26.1.320 C2802: Illegal character found: <Character>

[ERROR]

Description

In the source there is a character which does not match with the name rules for C/C++. As an example it is not legal to have '$' in identifiers. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
int $j;
```

Tips

Usually this is a programming error. Replace the illegal character with a legal one. Some E-MAIL programs set the most significant bit of two immediately following spaces. In a hex editor, such files then contain "a0 a0 a0 20" for four spaces instead of "20 20 20 20". When this occurs in your E-Mail configuration, send sources as attachment.

26.1.321 C2803: Limitation: Parser was going out of sync!

[ERROR]

Description

The parser was going out of synchronization. This is caused by complex code with many blocks, gotos and labels.

Example

```
It would take too much space to write an example here!
```

Tips

Try to simplify your code!

See also

- Limitations

26.1.322 C2900: Constant condition found, removing loop

[WARNING]

Description

A constant loop condition has been found and the loop is never executed. No code is produced for such a loop. Normally, such a constant loop condition may be a programming error.

Example

```
for(i=10; i<9; i--)



Because the loop condition 'i<9' never becomes true, the
loop is removed by the compiler and only the code for the
initialization 'i=10' is generated.
```

Tips

If it is a programming error, correct the loop condition.

See also

- Loop Unrolling

26.1.323 C2901: Unrolling loop

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A loop has been detected for unrolling. Either the loop has only one round, e.g.

```
for(i=1; i<2; i++)
```

or loop unrolling was explicitly requested (Compiler Option \c -Cu or <tt>#pragma LOOP_UNROLL</tt>) or the loop has been detected as empty as

```
for(i=1; i<200; i++);
```

Tips

If it is a programming error, correct the loop condition.

See also

- Loop Unrolling

26.1.324 C3000: File-stack-overflow (recursive include?)

[FATAL]

Description

There are more than 256 file includes in one chain or a possible recursion during an include sequence. Maybe the included header files are not guarded with ifndef

Example

```
// foo.c


#include "foo.c"
```

Tips

Use ifndef to break a possible recursion during include:

```
// foo.h


#ifndef FOO_H


#define FOO_H


// additional includes, declarations, ...


#endif
```

Simplify the include complexity to less than 256 include files in one include chain.

See also

- Limitations

26.1.325 C3100: Flag stack overflow -- flag ignored

[WARNING]

Description

There were too many flags used at the same time. This message occurs for Modula-2 versions of the compiler only. It does not occur for C/C++ compilers.

26.1.326 C3200: Source file too big

[FATAL]

Description

The compiler can handle about 400'000 lexical source tokens. A source token is either a number or an ident, e.g. 'int a[2] = {0,1};' contains the 12 tokens 'int', 'a', '[', '2', ']', '=', '{', '0', '1', '}' and ';'.

Example

A source file with more than 400'000 lexical tokens.

Tips

Split up the source file into parts with less then 400'000 lexical tokens.

See also

- Limitations

26.1.327 C3201: Carriage-Return without a Line-Feed was detected

[WARNING]

Description

On a PC, the usual 'newline' sequence is a carriage return (CR) followed by a line feed (LF). With this message the compiler warns that there is a CR without a LF. The reason could be a not correctly transformed UNIX source file. However, the compiler can handle correct UNIX source files (LF only). Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Tips

Maybe the source file is corrupted or the source file is not properly converted from a UNIX source file. Try to load the source file into an editor and to save the source file, because most of the editors will correct this.

26.1.328 C3202: Ident too long

[FATAL]

Description

An identifier was longer than allowed. The compiler supports identifiers with up to 16000 characters. Note that the limits of the linker/debugger may be smaller. The 16000 characters are in respect of the length of the identifier in the source. A name mangled C++ identifier is only limited by available memory.

Tips

Do not use such extremely large names!

26.1.329 C3300: String buffer overflow

[FATAL]

Description

The compiler can handle a maximum of about 10'000 strings in a compilation unit. If the compilation unit contains too many strings, this message will appear.

Example

```
A source file with more than 10'000 strings, e.g.
```

```
char *chPtr[] = {"string", "string1",


                 "string2", ... "string1000"};
```

Tips

Split up the source file into parts with less then 10'000 strings.

See also

- Limitations

26.1.330 C3301: Concatenated string too long

[FATAL]

Description

The compiler cannot handle an implicit string concatenation with a total of more than 8192 characters.

Example

Implicit string concatenation of two strings with each more than 4096 characters:

```
char *str = "MoreThan4096...."
"OtherWithMoreThan4096...."
```

Tips

Do not use implicit string concatenation, write the string in one piece:

```
char *str = "MoreThan4096....OtherWithMoreThan4096...."
```

See also

- Limitations
- C3303: Implicit concatenation of strings

26.1.331 C3302: Preprocessor-number buffer overflow

[FATAL]

Description

This message may occur during preprocessing if there are too many numbers to handle for the compiler in a compilation unit. The compiler can handle a most 10'000 different numbers for a compilation unit. Internally for each number a descriptor exists. If an internal number descriptor already exists for a given number value with a given type, the existing one is used. But if e.g. more than 10'000 different numbers are used, this message will appear.

Example

```
An array initialized with the full range of numbers from
0 to 10'000:



const int array[] = {0, 1, 2, ... 10000};
```

Tips

Splitting up the source file into smaller parts until this message disappears.

See also

- Limitations

26.1.332 C3303: Implicit concatenation of strings

[WARNING]

Description

ANSI-C allows the implicit concatenation of strings: Two strings are merged by the preprocessor if there is no other preprocessor token between them. This is a useful feature if two long strings should be one entity and you do want to write a very long line:

```
char *message = "This is my very long message string
which "
```

```
                "which has been splitted into two parts!"
```

This feature may be dangerous if there is just a missing comma (see example below!). If intention was to allocate a array of char pointers with two elements, the compiler only will allocate one pointer to the string "abcdef" instead two pointers if there is a comma between the two strings. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
char *str[] = {"abc" "def"}; // same as "abcdef"
```

Tips

If it is a programming error, correct it.

26.1.333 C3304: Too many internal ids, split up compilation unit

[FATAL]

Description

The compiler internally maintains some internal id's for artificial local variables. The number of such internal id's is limited to 256 for a single compilation unit.

Tips

Split up the compilation unit.

See also

- Limitations

26.1.334 C3400: Cannot initialize object (destination too small)

[ERROR]

Description

An object cannot been initialized, because the destination is too small, e.g. because the pointer is too small to hold the address. The message typically occurs if the programmer tries to initialize a near pointer (e.g. 16bit) with a far pointer (e.g. 24bit).

Example

```
#pragma DATA_SEG FAR MyFarSegment


char Array[10];


#pragma DATA_SEG DEFAULT


char *p = Array;
```

Tips

Increase the type size for the destination (e.g. with using the far keyword if supported)

```
char *far p = Array;
```

See also

- Limitations

26.1.335 C3401: Resulting string is not zero terminated

[WARNING]

Description

The compiler issues this message if a the resulting string is not terminated with a zero byte. Thus if such a string is used for printf or strcpy, the operation may fail. In C it is legal to initialize an array with a string if the string fits without the zero byte.

Example

```
void main(void) {
```

```
    char buf[3] = "abc";


}
```

Tips

For array initialization it is always better to use [] instead to specify the size:

```
char buf[] = "abc";
```

26.1.336 C3500: Not supported fixup-type for ELF-Output occurred

[FATAL]

Description

A fixup type not supported by the ELF Object file writer occurred. This message indicates an internal compiler error because all necessary fixup types must be supported. This message also can occur if in HLI (High Level Inline) Assembler an unsupported relocation/fixup type is used.

Tips

Report this error to your support.

26.1.337 C3501: ELF Error <Description>

[FATAL]

Description

The ELF generation module reports an error. Possible causes are when the object file to be generated is locked by another application or the disk is full.

Tips

Check if the output file exists and is locked. Close all applications which might lock it.

26.1.338 C3600: Function has no code: remove it!

[ERROR]

Description

It is an error when a function is really empty because such a function can not be represented in the memory and it does not have an address. Because all C functions have at least a return instruction, this error can only occur with the pragma NO_EXIT. Remark that not all targets support NO_EXIT.

Example

```
#pragma NO_EXIT


void main(void) {}
```

Tips

Remove the function. It is not possible to use an empty function.

26.1.339 C3601: Pragma TEST_CODE: mode <Mode>, size given <Size> expected <Size>, hashcode given <HashCode>, expected <HashCode>

[ERROR]

Description

The condition tested with a pragma TEST_CODE was false.

Example

```
#pragma TEST_CODE == 0


void main(void) {}
```

Tips

There are many reasons why the generated code may have been changed. Check why the pragma was added and which code is now generated. If the code is correct, adapt the pragma. Otherwise change the code.

See also

- pragma TEST_CODE

26.1.340 C3602: Global objects: <Number>, Data Size (RAM): <Size>, Const Data Size (ROM): <Size>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

This message is used to give some additional information about the last compilation unit compiled.

Example

Compile any C file.

Tips

Smart Linking may not link all variables or constants, so the real application may be smaller than this value. No alignment bytes are counted.

26.1.341 C3603: Static '<Function>' was not defined

[WARNING]

Description

A static function was used, but not defined. As static functions can only be defined in this compilation unit, the function using the undefined static cannot successfully link.

Example

```
static void f(void);
```

```
void main(void) {


  f();


}
```

Tips

Define the static function, remove its usage or declare it as external.

26.1.342 C3604: Static '<Object>' was not referenced

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A static object was defined but not referenced. When using smart linking, this object will not be used.

Example

```
static int i;
```

Tips

Remove the static object, comment it out or do not compile it with conditional compilation. Not referenced static functions often exist because the were used sometime ago but no longer or because the usage is present but not compiled because of conditional compilation.

26.1.343 C3605: Runtime object '<Object>' is used at PC <PC>

[**DISABLE** , INFORMATION, WARNING, ERROR]

Description

By default this message is disabled. This message may be enabled to report every runtime object used. Runtime objects (or calls) are used if the target CPU itself does not support a specific operation, e.g. a 32bit division or if the usage of such a runtime function is better than directly to inline it. The message is issued at the end of a function and reports the name and the PC where the object is used.

Example

```
double d1, d2;


void foo(void) {


  d1 = d2 * 4.5;  // compiler calls _DMUL for


                  // IEEE64 multiplication


}
```

Tips

Set this message to an error if you have to ensure that no runtime calls are made.

26.1.344 C3606: Initializing object '<Object>'

[**DISABLE** , INFORMATION, WARNING, ERROR]

Description

If global variables are initialized, such initialized objects have to be initialized during startup of the application. This message (which is disabled by default) reports every single global or static local initialization.

Example

```
int i = 3;       // message C3606
```

```
char *p = "abc"; // message C3606


void foo(void) {


  int k = 3;          // no message!


  static int j = 3;  // message C3606


}
```

Tips

Set this message to an error if you have to ensure that no additional global initialization is necessary for a copy down during startup of the application.

26.1.345 C3700: Special opcode too large

[FATAL]

Description

An internal buffer overflow in the ELF/DWARF 2 debug information output. This error indicates an internal error. It should not be possible to generate this error with legal input.

26.1.346 C3701: Too many attributes for DWARF2.0 Output

[FATAL]

Description

The ELF/DWARF 2 debug information output supports 128*128-128 different DWARF tags. This error indicates an internal error. It should not be possible to generate this error with legal input because similar objects use the same tag and there much less possible combinations.

26.1.347 C3800: Segment name already used

[ERROR]

Description

The same segment name was used for different segment type.

Example

```
#pragma DATA_SEG Test


#pragma CODE_SEG Test
```

Tips

Use different names for different types. If the two segments must be linked to the same area, this could be done in the link parameter file.

26.1.348 C3801: Segment already used with different attributes

[WARNING]

Description

A segment was used several times with different attributes.

Example

```
#pragma DATA_SEG FAR AA

..

#pragma DATA_SEG NEAR BB
```

. .

Tips

Use the same attributes with one segment. Keep variables of the same segment together to avoid inconsistencies.

26.1.349 C3802: Segment pragma incorrect

[WARNING]

Description

A section pragma was used with incorrect attributes or an incorrect name.

Example

```
#pragma DATA_SEG FAR FAR
```

Tips

Take care about not using keywords as names segment names. Note that you can use e.g. the __FAR_SEG instead FAR.

Example

```
#pragma DATA_SEG __FAR_SEG MyFarSeg
```

26.1.350 C3803: Illegal Segment Attribute

[WARNING]

Description

A Segment attribute was recognized, but this attribute is not applicable to this segment. Code segments may only be FAR, NEAR and SHORT. The DIRECT attribute is allowed for data segments only. The actual available segment attributes and their semantic depends on the target processor.

Example

```
#pragma CODE_SEG DIRECT MyFarSegment
```

Tips

Correct the attribute. Do not use segment attribute specifiers as segment names. Note that you can use the 'safe' qualifiers as well, e.g. __FAR_SEG.

26.1.351 C3804: Predefined segment '<segmentName>' used

[WARNING]

Description

A Segment name was recognized which is a predefined one. Predefined segment names are FUNCS, STRINGS, ROM_VAR, COPY, STARTUP, _PRESTART, SSTACK, DEFAULT_RAM, DEFAULT_ROM and _OVERLAP. If you use such segment names, this may raise conflicts during linking.

Example

```
#pragma CODE_SEG FUNCS   // WARNING here
```

Tips

Use another name. Do not use predefined segment names.

26.1.352 C3900: Return value too large

[ERROR]

Description

The return type of the function is too large for this compiler.

Example

```
typedef struct A {


  int i,j,k,l,m,n,o,p;


}A;


A f();
```

Tips

In C++, instead of a class/struct type, return a reference to it! In C, allocate the structure at the caller and pass a pointer/reference as additional parameter.

See also

- Compiler Backend

26.1.353 C4000: Condition always is TRUE

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler has detected a condition to be always true. This may also happen if the compiler uses high level optimizations, but could also be a hint for a possible programming error.

Example

```
unsigned int i= 2;


...


if (i >= 0) i = 1;
```

Example

```
extern void work(void);

void test(void) {

  while (1) {

    work();

  }

}
```

Tips

If it is a programming error, correct the statement. For endless loops, use for (;;) ... instead of while (1).

```
extern void work(void);

void test(void) {

  for (;;) {

    work();

  }

}
```

26.1.354 C4001: Condition always is FALSE

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler has detected a condition to be always false. This may also happen if the compiler uses high level optimizations, but could also be a hint for a possible programming error.

Example

```
unsigned int i;


if (-i < 0) i = -i;
```

Tips

If it is a programming error, correct the statement

26.1.355 C4002: Result not used

[WARNING]

Description

The result of an expression outside a condition is not used. In ANSI-C it is legal to write code as in the example below. Some programmers are using such a statement to enforce only a read access to a variable without write access, but in most cases the compiler will optimize such statements away.

Example

```
int i;


i+1; // should be 'i=1;', but programming error
```

Tips

If it is a programming error, correct the statement.

26.1.356 C4003: Shift count converted to unsigned char

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

In ANSI-C, if a shift count exceeds the number of bits of the value to be shifted, the result is undefined. Because there is no integral data type available with more than 256 bits yet, the compiler implicitly converts a shift count larger than 8 bits (char) to an unsigned char, avoiding loading a shift count too large for shifting, which does not affect the result. This ensures that the code is as compact as possible.

Example

```
int j, i;


i <<= j; // same as 'i <<= (unsigned char)j;'
```

In the above example, both 'i' and 'j' have type 'int', but the compile can safely replace the 'int' shift count 'j' with a 'unsigned char' type.

Tips

None, because it is a hint of compiler optimizations.

26.1.357 C4004: BitSet/BitClr bit number converted to unsigned char

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler has detected a shift count larger than 8bit used for a bitset/bitclear operation. Because it makes no sense to use a shift count larger than 256, the compiler optimizes the shift count to a character type. Reducing the shift count may reduce the code size and improve the code speed (e.g. a 32bit shift compared with a 8bit shift).

Example

```
int j; long L;


j |= (1<<L); // the compiler converts 'L'


             // to a unsigned character type
```

Tips

None, because it is a hint of compiler optimizations.

26.1.358 C4006: Expression too complex

[FATAL]

Description

The compiler cannot handle an expression which has more than 32 recursion levels.

Example

```
typedef struct S {


  struct S *n;


} S;


S *s;
```

```
void foo(void) {


  s->n-&gt;n->n-&gt;n-> ... n->n-&gt;n->n-&gt;n->n-
&gt;n->n-&gt;n = 0;


}
```

Tips

Try to simplify the expression, e.g. use temporary variables to hold expression results.

See also

- Limitations

26.1.359 C4007: Pointer deref is NOT allowed

[ERROR]

Description

The dereferencing operator cannot be applied. For some back-ends, pointer types that do not support this operation may exist (for instance, __linear pointers for HCS08).

26.1.360 C4100: Converted bit field signed -1 to 1 in comparison

[WARNING]

Description

A signed bitfield entry of size 1 can only have the values 0 and -1. The compiler did find a comparison of such a value with 1. The compiler did use -1 instead to generate the expected code.

Example

```
struct A {
```

```
  int i:1;


} a;


void f(void);


void main(void) {


  if (a.i == 1) {


    f();


  }


}
```

Tips

Correct the source code. Either use an unsigned bitfield entry or compare the value to -1.

26.1.361 C4101: Address of bitfield is illegal

[ERROR]

Description

The address of a bitfield was taken.

Example

```
typedef struct A {
```

```
  int bf1:1;


} A;


void f() {


  A a;


  if(&amp;a.bf1);


}
```

Tips

Use a "normal" integral member type, if you really need to have the address.

26.1.362 C4200: Other segment than in previous declaration

[WARNING]

Description

A object (variable or function) was declared with inconsistent segments.

Example

```
#pragma DATA_SEG A


extern int i;


#pragma DATA_SEG B
```

```
int i;
```

Tips

Change the segment pragmas in a way that all declarations and the definition of one object are in the same segment. Otherwise wrong optimizations could happen.

26.1.363 C4201: pragma <name> was not handled

[WARNING]

Description

A pragma was not used by the compiler. This may have different reasons: the pragma is intended for a different compiler by a typing mistake, the compiler did not recognize a pragma. Note that pragma names are case sensitive. there was no context, a specific pragma could take some action The segment pragma DATA_SEG, CODE_SEG, CONST_SEG and their aliases never issue this warning, even if they are not used.

Example

```
#pragma TRAP_PROG


// typing mistake: the interrupt pragma is called
TRAP_PROC


void Inter(void) {


  ...


}
```

Tips

Investigate this warning carefully. This warning can be mapped to an error if only pragmas are used which are known.

26.1.364 C4202: Invalid pragma OPTION,

[ERROR]

Description

A ill formed pragma OPTION was found or the given options were not valid. The description says more precisely what is the problem with a specific pragma OPTION.

Example

```
#pragma OPTION add "-or"
```

Tips

When the format was illegal, correct it. You can add comments, but they must follow the usual C rules. Be careful which options are given by the command line when adding options. It is not possible to add options which contradicts to command line options. Notice the limitations of the pragma OPTION.

See also

- pragma OPTION

26.1.365 C4203: Invalid pragma MESSAGE,

[WARNING]

Description

A ill formed pragma MESSAGE was found or the given message number cannot be moved. The description says more precisely what is the problem with a specific pragma MESSAGE.

Example

```
#pragma MESSAGE warning C4203
```

```
The pragma OPTION keyword warning is case sensitive!


Write instead:


#pragma MESSAGE WARNING C4203
```

Tips

When the format was illegal, correct it. You can add comments, but they must follow the usual C rules. The same message can be moved at different code positions to a different state. Be careful not to specify the same message with a option or with graphical user interface and with this pragma. If this is done, it is not defined which definition is actually taken.

See also

- pragma MESSAGE

26.1.366 C4204: Invalid pragma REALLOC_OBJ,

[ERROR]

Description

The pragma REALLOC_OBJ was used in a ill formed way.

See also

- pragma REALLOC_OBJ
- Linker Manual

26.1.367 C4205: Invalid pragma LINK_INFO,

[WARNING]

Description

The pragma LINK_INFO was used in a ill formed way.

See also

- pragma LINK_INFO

26.1.368 C4206: pragma pop found without corresponding pragma push

[ERROR]

Description

A pragma pop was found, but there was no corresponding pragma push.

See also

- pragma pop
- pragma push

26.1.369 C4207: Invalid pragma pop,

[WARNING]

Description

The pragma pop was used in a ill formed way.

See also

- pragma pop
- pragma push

26.1.370 C4208: Invalid pragma push,

[WARNING]

Description

The pragma push was used in a ill formed way.

See also

- pragma pop
- pragma push

26.1.371 C4209: Usage: pragma align (on|off),

[WARNING]

Description

The pragma align was not used together with on or off.

See also

- pragma align

26.1.372 C4300: Call of an empty function removed

[INFORMATION]

Description

If the option -Oi is enabled, calls of empty functions are removed.

Example

```
void f() {


}


void main() {


  f();        // this call is removed !


}
```

Tips

If for any reason you need a call of an empty function, disable the option -Oi.

See also

- Option -Oi

26.1.373 C4301: Inline expansion done for function call

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler was replacing the function call with the code of the function to be called.

Example

```
inline int f(int i) {

  return i+1;

}

void main() {

  int i=f(3);  // gets replaced by i=3+1;

}
```

Tips

To force the compiler to inline function calls use the keyword "inline".

See also

- Option -Oi

26.1.374 C4302: Could not generate inline expansion for this function call

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler could not replace the function call with the code of the function to be called. The expression containing the function call may be too complex, the function to be called may be recursive or too complex.

Example

```
inline int f(int i) {

  if(i>10) return 0;

  return f(i+1);

}

void main() {

  int i=f(3);  // Cannot inline,

               // because f() contains a recursive call

}
```

Tips

To have the same effect as inlining the function, replace the call with the code of the function manually.

26.1.375 C4303: Illegal pragma <name>

[WARNING]

Description

An parsing error was found inside of the given pragma.

Example

```
#pragma INLINE Inline it!


// pragma INLINE does not expect any arguments


void foo(void) {


}
```

Tips

- Check the exact definition of this pragma.
- Use comments to add text behind a pragma.

26.1.376 C4400: Comment not closed

[FATAL]

Description

The preprocessor has found a comment which has not been closed.

Example

```
/*
```

Tips

Close the comment.

26.1.377 C4401: Recursive comments not allowed

[WARNING]

Description

A recursive comment has been found (a comment with inside a comment).

Example

```
/* /* nested comment */
```

Tips

Either correct the comment, use

```
'#if 0'

...

'#endif'
```

or the C++ like comment '\\':

```
#if 0

  /* nested comment */

#endif

\\ /* /* nested comment */
```

26.1.378 C4402: Redefinition of existing macro '<MacroName>'

[FATAL]

Description

It is not allowed to redefine a macro.

Example

```
#define ABC 10


#define ABC 20
```

Tips

Correct the macro (e.g. using another name).

26.1.379 C4403: Macro-buffer overflow

[FATAL]

Description

There are more than 10'000 macros in a single compilation unit.

Example

```
#define macro0


#define macro1


...
```

```
#define macro10000
```

Tips

Simplify the compilation unit to reduce the amount of macro definitions.

See also

- Limitations

26.1.380 C4404: Macro parents not closed

[FATAL]

Description

In a usage of a macro with parameters, the closing parenthesis is not present.

Example

```
#define cat(a,b) (a##b)


int i = cat(12,34;
```

Tips

Add a closing ')'.

26.1.381 C4405: Include-directive followed by illegal symbol

[FATAL]

Description

After an include directive, there is an illegal symbol (not a file name in double quotes or within '<' and '>'.

Example

```
#include <string.h> <&lt;< message C4405 here
```

Tips

Correct the include directive.

26.1.382 C4406: Closing '>' missing

[FATAL]

Description

There is a missing closing '>' for the include directive.

Example

```
#include <string.h
```

Tips

Correct the include directive.

26.1.383 C4407: Illegal character in string or closing '>' missing

[FATAL]

Description

Either there is a non-printable character (as control characters) inside the file name for the include directive or the file name is not enclosed with '<' and '>'.

Example

```
#include <abc.h[control character]
```

Tips

If there are non-printable characters inside the file name, remove them. If there is a missing '>', add a '>' to the end of the file name.

26.1.384 C4408: Filename too long

[FATAL]

Description

A include file name longer than 2048 characters has been specified.

Example

```
#include <VeryLongFilename.....>
```

Tips

Shorten the file name, e.g. using a relative path or setting up the include file path in the default.env environment file.

See also

- Limitations

26.1.385 C4409: a ## b: the concatenation of a and b is not a legal symbol

[WARNING]

Description

The concatenation operator ## is used to concatenate symbols. If the resulting symbol is not a legal one, this message is issued. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define concat(a,b) a ## b


void foo(int a) {
```

```
    a concat(=,@) 5; // message: =@ is not a legal symbol


}
```

Tips

Check your macro definition. Generate a preprocessor output (option -Lp) to find the problem.

26.1.386 C4410: Unbalanced Parentheses

[FATAL]

Description

The number of opening parentheses '(' and the number of closing parentheses ')' does not match.

Tips

Check your macro definition. Generate a preprocessor output (option -Lp) to find the problem.

26.1.387 C4411: Maximum number of arguments for macro expansion reached

[FATAL]

Description

The compiler has reached the limit for the number of macro arguments for a macro invocation.

Example

```
#define A0(p1,p2,...,p1024) (p1+p2+...+p1024)
```

```
#define A1(p1,p2,...,p1024) A0(p1+p2+...+p1024)


void foo(void) {


  A1(1,2,...,1024); // message C4411 here


}
```

Tips

Try to avoid such a huge number of macro parameters, use simpler macros instead.

See also

- Limitations

26.1.388 C4412: Maximum macro expansion level reached

[FATAL]

Description

The compiler has reached the limit for recursive macro expansion. A recursive macro is if a macro depends on another macro. The compiler also stops macro expansion with this message if it seems to be an endless macro expansion.

Example

```
#define A0  0


#define A1  A0


#define A2  A1
```

...

Tips

Try to reduce huge dependency list of macros.

See also

- Limitations

26.1.389 C4413: Assertion: pos failed

[FATAL]

Description

This is a compiler internal error message only. It happens if during macro expansion the macro definition position is not the same as during the initial macro scanning.

Tips

If you encounter this message, please send us a preprocessor output (option -Lp).

26.1.390 C4414: Argument of macro expected

[FATAL]

Description

The preprocessor tries to resolve a macro expansion. However, there is no macro argument given after the comma separating the different macro arguments.

Example

```
#define Macro(a,b)


void foo(void) {
```

```
  Macro(,);


}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option -Lp) to find the problem.

26.1.391 C4415: ')' expected

[FATAL]

Description

The preprocessor expects a closing parenthesis. This may happen if a preprocessor macro has been called more argument than previously declared.

Example

```
#define A(a,b)  (a+b)


void main(void) {


  int i = A(3,4,5); // message C4415 here


}
```

Tips

Use the same number of arguments as declared in the macro.

26.1.392 C4416: Comma expected

[FATAL]

Description

The preprocessor expects a comma at the given position.

Tips

Check your macro definition or usage.

26.1.393 C4417: Mismatch number of formal, number of actual parameters

[FATAL]

Description

A preprocessor macro has been called with a different number of argument than previously declared.

Example

```
#define A(a,b)   (a+b)


void main(void) {


  int i = A(3,5,7); // message C4417 here


}
```

Tips

Use the same number of arguments as declared in the macro.

26.1.394 C4418: Illegal escape sequence

[ERROR]

Description

An illegal escape sequence occurred. A set of escape sequences is well defined in ANSI C. Additionally there are two forms of numerical escape sequences. The compiler has detected an escape sequence which is not covered by ANSI. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
char c= '\\p';
```

Tips

Remove the escape character if you just want the character. When this message is ignored, the compiler issued just the character without considering the escape character. So '\p' gives just a 'p' when this message is ignored. To specify and additional character use the either the octal or hexadecimal form of numerical escape sequences.

```
char c_space1= ' '; // conventional space


char c_space2= '\x20'; // space with hex notation


char c_space3= '\040'; // space with octal notation


\encode


Only 3 digits are read for octal numbers, so when you
specify 3 digits, then, there is no danger of combining
the numerical escape sequence with the next character in
sequence. Hexadecimal escape sequences should be
terminate explicitly.


\code


const char string1[]= " 0";// gives " 0"
```

```
const char string2[]= "\400";// error because 0400> 255


const char string3[]= "\x200";// error because 0x200 >
255


const char string4[]= "\0400";// gives " 0"


const char string5[]= "\x20"  "0";// gives " 0"
```

See also

- List of Escape Sequences

26.1.395 C4419: Closing ì missing

[FATAL]

Description

There is a string which is not terminated by a double quote. This message is also issued if the not closed string is at the end of the compilation unit file.

26.1.396 C4420: Illegal character in string or closing " missing

[FATAL]

Description

A string contains either an illegal character or is not terminated by a closing double quote.

Example

```
#define String "abc
```

Tips

Check your strings for illegal characters or if they are terminated by a double quote.

26.1.397 C4421: String too long

[FATAL]

Description

Strings in the preprocessor are actually limited to 8192 characters.

Tips

Use a smaller string or try to split it up.

See also

- Limitations

26.1.398 C4422: ' missing

[FATAL]

Description

To define a character, it has to be surrounded by single quotes (').

Example

```
#define CHAR 'a
```

Tips

Add a single quote at the end of the character constant.

26.1.399 C4423: Number too long

[FATAL]

Description

During preprocessing, the maximum length for a number has been reached. Actually this length is limited to 8192 characters.

Example

```
#define MyNum 12345......8193 // 8193 characters
```

Tips

Probably there is a typing error or the number is just too big.

See also

- Limitations

26.1.400 C4424: # in substitution list must be followed by name of formal parameter

[FATAL]

Description

There is a problem with the '#' operator during preprocessing, because there is no legal name as formal parameter specified.

Example

```
#define cat(a,b) a #


void foo(void) {


  i = cat(3,3);


}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option -Lp) to find the problem.

26.1.401 C4425: ## in substitution list must be preceded and followed by a symbol

[FATAL]

Description

There is a problem with the string concatenation ## operator during preprocessing, because there is no legal name as formal parameter specified.

Example

```
#define cat(a,b) a ##


void foo(void) {


  i = cat(3,3);


}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option -Lp) to find the problem.

26.1.402 C4426: Macro must be a name

[FATAL]

Description

There has to be a legal C/C++ ident to be used as a macro name. An ident has to be start with a normal letter (e.g. 'a'..'Z') or any legal ident symbol.

Example

```
#define "abc"
```

Tips

Use a legal macro name, e.g. not a string or a digit.

26.1.403 C4427: Parameter name expected

[FATAL]

Description

The preprocessor expects a name for preprocessor macros with parameters.

Example

```
#define A(3)  (B)
```

Tips

Do not use numbers or anything else than a name as macro parameter names.

26.1.404 C4428: Maximum macro arguments for declaration reached

[FATAL]

Description

The preprocessor has reached the maximum number of arguments allowed for a macro declaration.

Example

```
#define A(p0, p1, ..., p1023, p1024) (p0+p1+...p1024)
```

Tips

Try to split your macro into two smaller ones.

See also

- Limitations

26.1.405 C4429: Macro name expected

[FATAL]

Description

The preprocessor expects macro name for the undef directive.

Example

```
#undef #xyz
```

Tips

Use only a legal macro name for the undef directive.

26.1.406 C4430: Include macro does not expand to string

[FATAL]

Description

The file name specified is not a legal string. A legal file name has to be surrounded with double quotes "".

Example

```
#define file 3


#include file
```

Tips

Specify a legal file name.

26.1.407 C4431: Include "filename" expected

[FATAL]

Description

There is no legal file name for a include directive specified. A file name has to be non-empty and surrounded either by '<' and '>' or by double quotes "".

Example

```
#include <>
```

Tips

Specify a legal file name.

26.1.408 C4432: Macro expects '('

[FATAL]

Description

While expanding macros, the preprocessor expects here an opening parenthesis to continue with macro expansion.

Tips

Check your macro definition or usage. Generate a preprocessor output (option -Lp) to find the problem.

26.1.409 C4433: Defined <name> expected

[FATAL]

Description

Using 'defined', it can be checked if a macro is defined or not. However, there has to be a name used as argument for defined.

Example

```
#if defined()


#endif
```

Tips

Specify a name for the defined directive, e.g. if defined(abc).

26.1.410 C4434: Closing ')' missing

[FATAL]

Description

During macro expansion, the preprocessor expects a closing parenthesis to continue.

Tips

Check your macro definition or usage. Generate a preprocessor output (option -Lp) to find the problem.

26.1.411 C4435: Illegal expression in conditional expression

[FATAL]

Description

There is an illegal conditional expression used in a if or elif directive.

Example

```
#if (3*)
```

```
#endif
```

Tips

Check the conditional expression.

26.1.412 C4436: Name expected

[FATAL]

Description

The preprocessor expects a name for the ifdef and ifndef directive.

Example

```
#ifndef 3333_H // <&lt; C4436 here


#define 3333_H


#endif
```

Tips

Check if a legal name is used, e.g. it is not legal to start a name with a digit.

26.1.413 C4437: Error-directive found: <message>

[ERROR]

Description

The preprocessor stops with this message if he encounters an error directive. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#error "error directive"
```

Tips

Check why the preprocessor evaluates to this error directive. Maybe you have forgotten to define a macro which has caused this error directive.

26.1.414 C4438: Endif-directive missing

[FATAL]

Description

All if or ifdef directives need a endif at the end. If the compiler does not find one, this message is issued.

Example

```
#if 1
```

Tips

Check where the endif is missing. Generate a preprocessor output (option -Lp) to find the problem.

26.1.415 C4439: Source file <file> not found

[FATAL]

Description

The compiler did not found the source file to be used for preprocessing.

Tips

Check why the compiler was not able to open the indicated file. Maybe the file is not accessible any more or locked by another application.

26.1.416 C4440: Unknown directive: <directive>

[FATAL]

Description

The preprocessor has detected a directive which is unknown and which cannot be handled.

Example

```
#notadirective
```

Tips

Check the directive. Maybe it is a non-ANSI directive supported by another compiler.

26.1.417 C4441: Preprocessor output file <file> could not be opened

[FATAL]

Description

The compiler was not able to open the preprocessor output file. The preprocessor file is generated if the option -Lp is specified.

Tips

Check your macro definition or usage. Check the option -Lp: the format specified may be illegal.

26.1.418 C4442: Endif-directive missing

[FATAL]

Description

The asm directive needs a endasm at the end. If the compiler does not find one, this message is issued.

Example

```
#asm
```

Tips

Check where the endasm is missing. Generate a preprocessor output (option -Lp) to find the problem.

26.1.419 C4443: Undefined Macro 'MacroName' is taken as 0

[WARNING]

Description

Whenever the preprocessor evaluates a condition and finds a identifier which was not defined before, he implicitly takes its value to be the integral value 0. This C style behavior may arise in hard to find bug. So when the header file, which actually defines the value is not included or when the macro name was entered incorrectly, for example with a different case, then the preprocessor "#if" and "#elif" instructions wont behave as expected. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define _Debug_Me 1


...


void main(int i) {


#if Debug_Me // missing _ in front of _Debug_Me.


  assert(i!=0);
```

```
#endif


}
```

The assert will never be reached.

Tips

This warning is a good candidate to be mapped to an error. To save test macros defined in some header files, also test if the macros are defined:

```
#define _Debug_Me


...


void main(int i) {


#ifndef(Debug_Me)


#error // macro must be defined above


#endif


#if Debug_Me // missing _ in front of _Debug_Me.


  assert(i!=0);


#endif


}
```

The checking of macros with "#ifdef" and "#ifndef" cannot detect if the header file, a macro should define is really included or not. Note that using a undefined macro in C source will treat the macro as C identifier and so usually be remarked by the compiler. Also note a undefined macro has the value 0 inside of preprocessor conditions, while a defined macro with nothing as second argument of a "#define" replaces to nothing. E.g.

```
#define DEFINED_MACRO



#if UNDEFINED_MACRO // evaluates to 0, giving this
warning



#endif



if DEFINED_MACRO   // error FATAL: C4435: Illegal



                   // expression in conditional expression



#endif
```

26.1.420 C4444: Line number for line directive must be > 0 and <= 32767

[WARNING]

Description

ANSI requires that the line number for the line directive is greater zero or smaller-equal than 32767. If this message occurs and it is currently mapped to an error, the compiler sets the line number to 1.

Example

```
#line 0
```

```
#line 32768 "myFile.c"
```

Tips

Specify a line number greater zero and smaller 32768. For automatically generated code, which has such illegal line directives, you can move this error to a warning.

26.1.421 C4445: Line number for line directive expected

[ERROR]

Description

ANSI requires that after the line directive a number has to follow.

Example

```
#line // <&lt; ERROR C4445


#line "myFile.c" // <&lt; ERROR C4445
```

Tips

Specify a line number greater zero and smaller 32768.

26.1.422 C4446: Missing macro argument(s)

[WARNING]

Description

In a macro 'call', one or more arguments are missing. The pre-processor replaces the parameter string with nothing. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define f(A, B) int A B


void main(void){


  f(i,);// this statement will be replaced with 'int i;'


        // by the pre-processor.


}
```

Tips

Be careful with empty macro arguments, because the behavior is undefined in ANSI-C. So avoid it if possible.

26.1.423 C4447: Unexpected tokens following preprocessor directive - expected a newline

[WARNING]

Description

The compiler has detected that after a directive there was something unexpected. Directives are normally line oriented, thus the unexpected tokens are just ignored. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#include "myheader.h" something
```

Tips

Remove the unexpected tokens.

26.1.424 C4448: Warning-directive found: <message>

[WARNING]

Description

The preprocessor stops with this message if he encounters an warning directive. Note that this directive is only support if -Ansi is not set. Note: The pragma MESSAGE does not apply to this message because it is issued in the preprocessing phase.

Example

```
#warning "warning directive"
```

Tips

Check why the preprocessor evaluates to this warning directive. Maybe you have forgotten to define a macro which has caused this directive.

26.1.425 C4449: Exceeded preprocessor if level of 4092

[ERROR]

Description

The preprocessor does by default only allow 4092 concurrently open if directives. If more do happen, this message is printed. Usually this message does only happen because of a programming error.

Example

```
#if 1 // 0


#if 2 // 0


#if 3 // 0
```

```
.....


#if 4092 // 0
```

Tips

Check why there are that many open preprocessor if's. Are the endif's missing?

26.1.426 C4450: Multi-character character constant

[WARNING]

Description

This warning is issued by the compiler to indicate that the source code contains a multi-character character constant.

Example

```
char var = `12' //Error
```

26.1.427 C4700: Illegal pragma TEST_ERROR

[WARNING]

Description

The pragma TEST_ERROR is for internal use only. It is used to test the message management and also to test error cases in the compiler.

26.1.428 C4701: pragma TEST_ERROR: Message <ErrorNumber> did not occur

[ERROR]

Description

The pragma TEST_ERROR is for internal use only. It is used to test the message management and also to test error cases in the compiler.

26.1.429 C4800: Implicit cast in assignment

[WARNING]

Description

An assignment requiring an implicit cast was made.

Tips

Check, if the casting results in correct behavior.

26.1.430 C4801: Too many initializers

[ERROR]

Description

The braced initialization has too many members.

Example

```
char name[4]={'n','a','m','e',0};
```

Tips

Write the correct number of members in the braced initializer.

26.1.431 C4802: String-initializer too large

[ERROR]

Description

The string initializer was too large.

Tips

Take a shorter string, or try to allocate memory for your string in an initialization function of the compilation unit.

26.1.432 C4900: Function differs in return type only

[ERROR]

Description

The overloaded function has the same parameters, but not the same return type.

Example

```
void f(int);


void f();


int f(int);  // error
```

Tips

A function redeclaration with the same parameters must have the same return type than the first declaration.

26.1.433 C5000: Following condition fails: sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

[ERROR]

Description

Type sizes has been set to illegal sizes. For compliance with the ANSI-C rules, the type sizes of char, short, int, long and long long has to in a increasing order, e.g. setting char to a size of two and int to a size of one will violate this ANSI rule.

Example

```
-Tc2i1
```

Tips

Change the -T option.

26.1.434 C5001: Following condition fails: sizeof(float) <= sizeof(double) <= sizeof(long double) <= sizeof(long long double)

[ERROR]

Description

Your settings of the Standard Types are wrong!

Example

```
-Tf4d2
```

Tips

Set your Standard Types correctly

See also

- Change the -T option.

26.1.435 C5002: Illegal type

[ERROR]

Description

A unknown or illegal type occurred. This error may happen as consequence of another error creating the illegal type.

Tips

Check for other errors happening before.

26.1.436 C5003: Unknown array-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

26.1.437 C5004: Unknown struct-union-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

26.1.438 C5005: PACE illegal type

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

26.1.439 C5006: Illegal type settings for HIWARE Object File Format

[ERROR]

Description

For HIWARE object file format (option -Fh, -F7 and default if no other object file format is selected) the char type must always be of size 1. This limitation is because there may not be any new types introduced in this format and 1 byte types are used internally in the compiler ever if the user only need multibyte characters. For the strict HIWARE object file format (option -F7) the additional limitation that the enum type has the size 2 bytes, and must be signed, is checked with this message. The HIWARE Object File Format (-Fh) has following limitations: The type char is limited to a size of 1 byte Symbolic debugging for enumerations is limited to 16bit signed enumerations No symbolic debugging for enumerations No zero bytes in strings allowed (zero byte marks the end of the string) The strict HIWARE V2.7 Object File Format (option -F7) has some limitations: The type char is limited to a size of 1 byte Enumerations are limited to a size of 2 and has to be signed No symbolic debugging for enumerations The standard type 'short' is encoded as 'int' in the object file format No zero bytes in strings allowed (zero byte marks the end of the string)

Example

```
COMPOPTIONS= \c -Te2 \c -Fh
```

Tips

Use -Fh HIWARE object file format to change the enum type. To change the char type, only the ELF object file format can be used (if supported). Note that not all backends allow the change of all types.

See also

- Option for object file format -Fh, -F7, -F1, -F2
- Option for Type Setting -T

26.1.440 C5100: Code size too large

[ERROR]

Description

The code size is too large for this compiler.

Example

```
// a large source file
```

Tips

Split up your source into several compilation units!

See also

- Limitations

26.1.441 C5200: 'FileName' file not found

[ERROR]

Description

The specified source file was not found.

Example

```
#include "notexisting.h"
```

Tips

Specify the correct path and name of your source file!

See also

- Input Files

26.1.442 C5250: Error in type settings: <Msg>

[ERROR]

Description

There is an inconsistent state in the type option settings. E.g. it is illegal to have the 'char' size larger than the size for the type 'short'.

Tips

Check the -T option in your configuration files. Check if the option is valid.

See also

- Option -T

26.1.443 C5300: Limitation: code size '<actualSize>' > '<limitSize>' bytes

[ERROR]

Description

You have a limited version of the compiler or reached the limitation specified in the license file. The actual demo limitation is 1024 bytes of code for 8/16bit targets and 3KByte for 32bit targets (without a license file). Depending on the license configuration, the code size limit may be specified in the license file too.

Tips

Check if you have enough licenses if you are using a floating license configuration. Check for the correct location of the license file. Get a license for a full version of the compiler, or for a code size upgrade.

26.1.444 C5302: Couldn't open the object file <Descr>

[FATAL]

Description

The compiler cannot open the object file for writing.

Tips

Check if there is already an object file with the same name but used by another application. Check if the object file is marked as read-only or locked by another application. Check if the output path does actually exist.

26.1.445 C5320: Cannot open logfile '<FileName>'

[FATAL]

Description

The compiler cannot open the logfile file for writing.

Tips

Check if there is already a file with the same name but used by another application. Check if the file is marked as read-only or locked by another application.

See also

- Option -Ll

26.1.446 C5350: Wrong or invalid encrypted file '<File>' (<MagicValue>)

[FATAL]

Description

The compiler cannot read an encrypted file because the encrypted file magic value is wrong.

Tips

Check if the file is a valid encrypted file.

See also

- Option -Eencrypt
- Option -Ekey

26.1.447 C5351: Wrong encryption file version: '<File>' (<Version>)

[FATAL]

Description

The compiler cannot read the encrypted file because the encryption version does not match.

Tips

Check if you have a valid license for the given encryption version. Check if you use the same license configuration for encryption and encrypted file usage.

See also

- Option -Eencrypt
- Option -Ekey

26.1.448 C5352: Cannot build encryption destination file: '<FileSpec>'

[FATAL]

Description

Building the encryption destination file name using the 'FileSpec' was not possible.

Tips

Check your FileSpec if it is legal.

See also

- Option -Eencrypt
- Option -Ekey

26.1.449 C5353: Cannot open encryption source file: '<File>'

[FATAL]

Description

It was not possible to open the encryption source file.

Tips

Check if the source file exists.

See also

- Option -Eencrypt
- Option -Ekey

26.1.450 C5354: Cannot open encryption destination file: '<File>'

[FATAL]

Description

The compiler was not able to write to the encryption destination file.

Tips

Check if you have read/write access to the destination file. Check if the destination file name is a valid one. Check if the destination file is locked by another application.

See also

- Option -Eencrypt
- Option -Ekey

26.1.451 C5355: Encryption source '<SrcFile>' and destination file '<DstFile>' are the same

[FATAL]

Description

The encryption source file and the destination file are the same. Because it is not possible to overwrite the source file with the destination file, encryption is aborted.

Tips

Change the encryption destination file name specification.

See also

- Option -Eencrypt
- Option -Ekey

26.1.452 C5356: No valid license for encryption support

[FATAL]

Description

It was not possible to check out the license for encryption support.

Tips

Check your license configuration. Check if you have a valid encryption license.

See also

- Option -Lic

26.1.453 C5400: Internal data structure inconsistency (SSA)

[ERROR]

Description

The internal data structure (Static Single Assignment form) of the compiler is wrong. Please extract the problem and send it to support. This error can be converted to a warning, because in most cases the inconsistency does not lead to wrong target code.

Example

```
This is a internal error which should not occur.
```

Tips

Try to find a different formulation as workaround. Contact support about this problem.

See also

- Limitations

26.1.454 C5401: Internal optimized data structure inconsistency (SSA)

[ERROR]

Description

The optimized internal data structure (Static Single Assignment form) of the compiler is wrong. Please extract the problem and send it to your support. This error can be converted to a warning, because in most cases the inconsistency does not lead to wrong target code. This is a internal error which should not occur.

Tips

Try to find a different formulation as workaround.

See also

- Limitations

26.1.455 C5403: Trying to take address of register

[ERROR]

Description

The compiler tries to take the address of a register which is not possible.

Example

```
void main(void) {


  int a, b;


  int *p;
```

```
    p = &amp;(a+b);  (ERROR: result a+b is in register)


}
```

Tips

Do not use the address operator for temporary results.

See also

- Limitations

26.1.456 C5500: Incompatible pointer operation

[WARNING]

Description

Two pointers of different sizes are subtracted (pointer difference).

Example

```
void main(void) {


  int *p;


  int *far q;






  *(p-q) = 0;  (Error)


}
```

Tips

Cast one of the pointer to get equal size.

See also

- Limitations

26.1.457 C5650: Too many locations, try to split up function

[FATAL]

Description

The internal data structure of the compiler blows up. One of your functions is too large.

Tips

Split up the function that causes the message.

See also

- Limitations

26.1.458 C5651: Local variable <variable> may be not initialized

[WARNING]

Description

The compiler issues this warning if no dominant definition is found for a local variable when it is referenced. Ignore this message if the local variable is defined implicitly (e.g. by inline assembly statements).

Example

```
int f(int param_i) {


  int local_i;
```

```
    if(param_i  == 0) {


       local_i = 1;    // this definition for local_i does


                       // NOT dominate following usage


    }


    return local_i;   // local_i referenced: no dominant


                      // definition for local_i found


  }
```

Tips

Review the code and initialize local variables at their declaration (e.g. local_i = 0).

26.1.459 C5660: Removed dead code

[WARNING]

Description

The compiler has optimized some unused code away.

Tips

Sometimes these compiler message shows some problem in the C code.

26.1.460 C5700: Internal Error <ErrorNumber> in '<Module>', please report to <Producer>

[FATAL]

Description

The Compiler is in an internal error state. Please report this type of error as described in the chapter Bug Report This message is used while the compiler is not investigating a specific function.

Example

no example known.

Tips

Sometimes these compiler bugs occur in wrong C Code. So look in your code for incorrect statements. Simplify the code as long as the bug exists. With a simpler example, it is often clear what is going wrong and how to avoid this situation. Try to avoid compiler optimization by using the volatile keyword. Please report errors for which you do have a work around.

See also

- Chapter Bug Report

26.1.461 C5701: Internal Error #<ErrorNumber> in '<Module>' while compiling file '<File>, procedure '<Function>', please report to <Producer>

[FATAL]

Description

The Compiler is in an internal error state. Please report this type of error as described in the chapter Bug Report. This message is used while the compiler is investigating a specific function.

Example

no example known.

Tips

Sometimes these compiler bugs occur in wrong C Code. So look in your code for incorrect statements. Simplify the code as long as the bug exists. With a simpler example, it is often clear what is going wrong and how to avoid this situation. Try to avoid compiler optimization by using the volatile keyword. Please report errors for which you do have a work around.

See also

- Chapter Bug Report

26.1.462 C5702: Local variable '<Variable>' declared in function '<Function>' but not referenced

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected a local variable which is not used.

Example

```
void main(void) {


  int i;


}
```

Tips

Remove the variable if it is never used. If it is used in some situations with conditional compilation, use the same conditions in the declaration as in the usages.

26.1.463 C5703: Parameter '<Parameter>' declared in function '<Function>' but not referenced

[INFORMATION]

Description

The Compiler has detected a named parameter which is not used. In C parameters in function definitions must have names. In C++ parameters may have a name. If it has no name, this warning is not issued. This warning may occur in cases where the interface of a function is given, but not all parameters of all functions are really used.

Example

```
void main(int i) {


}
```

Tips

If you are using C++, remove the name in the parameter list. If you are using C, use a name which makes clear that this parameter is intentionally not used as, for example "dummy".

26.1.464 C5800: User requested stop

[ERROR]

Description

This message is used when the user presses the stop button in the graphical user interface. Also when the compiler is closed during a compilation, this message is issued.

Tips

By moving this message to a warning or less, the stop functionality can be disabled.

26.1.465 C5900: Result is zero

[WARNING]

Description

The Compiler has detected operation which results in zero and is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j-j;  // optimized to i = 0;
```

Tips

If it is a programming error, correct the statement.

26.1.466 C5901: Result is one

[WARNING]

Description

The Compiler has detected an operation which results in one. This operation is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j/j;  // optimized to i = 1;
```

Tips

If it is a programming error, correct the statement.

26.1.467 C5902: Shift count is zero

[WARNING]

Description

The Compiler has detected an operation which results in a shift count of zero. The operation is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j<&lt;(j-j);  // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

26.1.468 C5903: Zero modulus

[WARNING]

Description

The Compiler has detected a % operation with zero. Because the modulus operation implies also a division (division by zero), the compiler issues a warning. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j%0;  // error
```

Tips

Correct the statement.

26.1.469 C5904: Division by one

[WARNING]

Description

The Compiler has detected a division by one which is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j/1;  // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

26.1.470 C5905: Multiplication with one

[WARNING]

Description

The Compiler has detected a multiplication with one which is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j*1;  // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

26.1.471 C5906: Subtraction with zero

[WARNING]

Description

The Compiler has detected a subtraction with zero which is optimized. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j-(j-j);  // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

26.1.472 C5907: Addition replaced with shift

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected a addition with same left and right expression which is optimized and replaced with a shift operation. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
i = j+j;  // optimized to i = j<<1;
```

Tips

If it is a programming error, correct the statement.

26.1.473 C5908: Constant switch expression

[WARNING]

Description

The Compiler has detected a constant switch expression. The compiler optimizes and reduces such a switch expression This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
switch(2){

  case 1: break;

  case 2: i = 0; break;

  case 3: i = 7; break;

};  // optimized to i = 0;
```

Tips

If it is a programming error, correct the statement.

26.1.474 C5909: Assignment in condition

[WARNING]

Description

The Compiler has detected an assignment in a condition. Such an assignment may result from a missing '=' which is normally a programming error. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (i = 0)  // should be 'i == 0';
```

Tips

If it is a programming error, correct the statement.

26.1.475 C5910: Label removed

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected a label which can be optimized . This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
switch(i) {


  Label: i = 0;  // Labeled removed


   ...
```

```
if (cond) {

  L2:  // L2 not referenced: Label removed

  ...

} else {

}
```

Tips

Do not use normal labels in switch statements. If it is a switch case label, do not forget to add the 'case' keyword.

26.1.476 C5911: Division by zero at runtime

[WARNING]

Description

The Compiler has detected zero division. This is not necessarily an error (see below). This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
void RaiseDivByZero(void) {

  int i = i/0;  // Division by zero!

}
```

Tips

Maybe the zero value the divisor results from other compiler optimizations or because a macro evaluates to zero. It is a good idea to map this warning to an error (see Option -WmsgSe).

26.1.477 C5912: Code in 'if' and 'else' part are the same

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected that the code in the 'if' and the code in the 'else' part of an 'if-else' construct is the same. Because regardless of the condition in the 'if' part, the executed code is the same, so the compiler replaces the condition with 'TRUE' if the condition does not have any side effects. There is always a couple of this message, one for the 'if' part and one for the 'else' part. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (condition) {  // replaced with 'if (1) {'

  statements;  // message C5912 here ...

} else {

  statements;  // ... and here

}
```

Tips

Check your code why both parts are the same. Maybe different macros are used in both parts which evaluates to the same values.

26.1.478 C5913: Conditions of 'if' and 'else if' are the same

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected that the condition in an 'if' and a following 'else if' expression are the same. If the first condition is true, the second one is never evaluated. If the first one is FALSE, the second one is useless, so the compiler replaces the second condition with 'FALSE' if the condition does not have any side effects. There is always a couple of this message, one for the 'if' part and one for the 'if else' part. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (condition) { // message C5913 here

  ...;

} else if(condition) { // here, condition replaced with 0

  ...

}
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both conditions which evaluates to the same values.

26.1.479 C5914: Conditions of 'if' and 'else if' are inverted

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected that the condition in an 'if' and a following 'else if' expression are just inverted. If the first condition is true, the second one is never evaluated (FALSE). If the first one is FALSE, the second one is TRUE, so the compiler replaces the second condition with 'TRUE' if the condition does not have any side effects. There is always a couple of this message, one for the 'if' condition and one for the 'if else' condition. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (condition) {  // message C5914 here ...

  ...;

} else if(!condition) {// here, condition replaced with 1

  ...

}
```

Tips

Check your code why both conditions are inverted. Maybe different macros are used in both parts which evaluates to the same values.

26.1.480 C5915: Nested 'if' with same conditions

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected that the condition in an 'if' and a nested 'if' expression have the same condition. If the first condition is true, the second one is always true. If the first one is FALSE, the second one is FALSE too, so the compiler replaces the second condition with 'TRUE' if the condition does not have any side effects. There is always a couple of this message, one for the first 'if' condition and one for nested 'if' condition. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (condition) {  // message C5915 here ...

  if(!condition) { // here, condition replaced with 1

  ...

}
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both parts which evaluates to the same values.

26.1.481 C5916: Nested 'if' with inverse conditions

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The Compiler has detected that the condition in an 'if' and a nested 'if' expression have the inverse condition. If the first condition is true, the second one is always false. If the first one is FALSE, the second one is TRUE, so the compiler replaces the second condition with 'FALSE' if the condition does not have any side effects. There is always a couple of this message, one for the first 'if' condition and one for nested 'if' condition. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
if (condition) {  // message C5916 here ...

  if(!condition) { // here, condition replaced with 0
```

```
    ...


}
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both parts which evaluates to the same values.

26.1.482 C5917: Removed dead assignment

[WARNING]

Description

The Compiler has detected that there is an assignment to a (local) variable which is not used afterwards. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
    int a;

    ...

    a = 3 // message C5917 here ...


} // end of function
```

Tips

If you want to avoid this optimization, you can declare the variable as volatile.

26.1.483 C5918: Removed dead goto

[WARNING]

Description

The Compiler has detected that there is goto jumping to a just following label. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
goto Label; // message C5918 here ...


Label:


...
```

Tips

If you want to avoid this optimization, you can declare the variable as volatile.

26.1.484 C5919: Conversion of floating to unsigned integral

[WARNING]

Description

In ANSI-C the result of a conversion operation of a (signed) floating type to a unsigned integral type is undefined. One implementation may return 0, another the maximum value of the unsigned integral type or even something else. Because such behavior may cause porting problems to other compilers, a warning message is issued for this.

Example

```
float f = -2.0;


unsigned long uL = f; // message C5919 here
```

Tips

To avoid the undefined behavior, first assign/cast the floating type to a signed integral type and then to a unsigned integral type.

See also

- ISO/IEC 9899:1990 (E), page 35, chapter 6.2.1.3 Floating and integral: "When a value of floating type is converted to integral type, the fractional part is discarded. The value of the integral part cannot be represented by the integral type, the behavior is undefined."

26.1.485 C5920: Optimize library function <function>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

The compiler has optimized the indicated library function. Depending on the actual function and its arguments, the compiler does replace the function with a simpler one or does even replace the function with the actual code as if the ANSI-C function would have been called. If you want to your a certain actual implementation of this function, disable this optimization with the -oilib option. There is a certain -oilib suboption for every supported ANSI-C function.

Seealso

- Option -OiLib

26.1.486 C5921: Shift count out of range

[WARNING]

Description

The compiler has detected that there is a shift count exceeding the object size of the object to be shifted. This is normally not a problem, but can be optimized by the compiler. For right shifts (>>), the compiler will replace the shift count with (sizeOfObjectInBits-1), that is a shift of a 16bit object (e.g. a short value) with a right shift by twenty is replaced with a right shift by 15. This message may be generated during tree optimizations (Option -Ont to switch it off).

Example

```
unsigned char uch, res;


res = uch >&gt; 9;  // uch only has 8 bits, warning here


                // will be optimized to 'res = uch>&gt;7'
```

See also

- Option -Ont

26.1.487 C6000: Creating Asm Include File <file>

[DISABLE, **INFORMATION** , WARNING, ERROR]

Description

A new file was created containing assembler directives. This file can be included into any assembler file to automatically get information from C header files.

Tips

Use this feature when you have both assembler and C code in your project.

See also

- Option -La
- Create Assembler Include Files
- pragma CREATE_ASM_LISTING

26.1.488 C6001: Could not Open Asm Include File because of <reason>

[WARNING]

Description

The Assembler Include file could not be opened. As reason either occurs the file name, which was tried to open or another description of the problem.

Tips

Try to specify the name directly. Check the usage of the file name modifiers. Check of the file exists and is locked by another application

See also

- Option -La
- Create Assembler Include Files
- pragma CREATE_ASM_LISTING

26.1.489 C6002: Illegal pragma CREATE_ASM_LISTING because of <reason>

[ERROR]

Description

The pragma CREATE_ASM_LISTING was used in a ill formed way.

Tips

After the pragma, the may only be a ON or OFF. ON and OFF are case sensitive and must not be surrounded by double quotes.

Example

```
#pragma CREATE_ASM_LISTING ON
```

See also

- Option -La
- Create Assembler Include Files
- pragma CREATE_ASM_LISTING

26.1.490 Messages of HC08 Back End

Messages of HC08 Back End The following sections contains message descriptions specific for the HC08 compiler. All messages specific for the HC08 have a number of the form C18XXX, where XXX is a different number for every message. Up to the compiler version 5.0.7, there is also a set of messages with numbers C20XXX (ICG based technology).

26.1.491 C18000: Label not set

[ERROR]

Description

The HLI assembler assumes every unknown symbol to be a label. But if this symbol/label is not defined, this message will occur.

Example

```
asm lda xyz;  // error
```

Tips

Check if this label exists or if there is a misspelling. If it is a C/C++ object, check if there is a declaration of it.

26.1.492 C18001: Incompatible memory model (banked memory model) for the chosen derivative

[ERROR]

Description

The derivative you generate code for does not support code banking.

Tips

Chose a derivative that supports code banking or change the memory model to small or tiny.

26.1.493 C18002: Pointer conversion not supported

[ERROR]

Description

An unsupported pointer conversion was encountered. For some back-ends, such unsupported conversions may exist (between pointers qualified with non-ANSI keywords).

26.1.494 C18003: _linear pointer to object in non-LINEAR CONST_SEG [Object addresses extended]

[WARNING

Description

A linear pointer is initialized with the address of an object defined in a segment that is not qualified with __LINEAR_SEG.

Example

```
#pragma CONST_SEG my_const_seg


const int i;


const int * __linear p = &i; //C18003 occurs here.
```

26.1.495 C18100: Number expected

[ERROR]

Description

The HLI assembler expects a number. This is either constant expression or a single number.

Example

```
asm lda (3*i),i;  // error
```

Tips

Do not use objects in constant expressions.

26.1.496 C18004: MMU can be used for HCS08 derivatives only

ERROR]

Description

This error message is issued if the -MMU option is used without the -Cs08 option when passing arguments to the compiler.

Example

```
chc08.exe-MMU             /* Error */


chc08.exe-MMU -Cs08  /*Correct use of MMU option*/
```

26.1.497 C18005: Unsupported pointer qualifier combination for function pointer

[ERROR]

Description

A function pointer may have a qualifier that specifies the calling convention and one that specifies the pointer size. For example, consider the following function pointer: int __far (* __far f)(void). The leftmost qualifier specifies the calling convention, while the other qualifier indicates the pointer size. The HC(S)08 backend only accepts far pointers to far functions and near pointers to near functions. Any other combination (e.g. int __far (*

__near f)(void)) will result in a compiler error. The compiler also checks the memory model (on banked memory model, all functions are __far by default if not specified otherwise).

Example

```
int __far (* f)(void); // legal if compiling for banked
memory model, illegal for small or tiny.
```

Tips

Do not use near pointers to far functions or far pointers to near functions.

26.1.498 C18101: Object is not a field

ERROR]

Description

The HLI assembler expects a structure's field or a class member on the right side of the "." assembly operator.

Example

```
struct _str { int flag;}


_str port;


asm LDA  port.field


 /* Error, field is not a member of the _str struct. */


asm LDA port.flag


/*correct use*/
```

26.1.499 C18102: Object is not a structure

ERROR]

Description

The HLI assembler expects a struct / class type on the left side of the "." assembly operator.

Example

```
struct _str { int flag;}


int port;


asm LDA  port.flag


 /*Error, port is not of struct type*/
```

26.1.500 C18103: Factor expected

[ERROR]

Description

The HLI assembler expects a factor. A factor is a sequence starting with

- left parenthesis '('
- cross '#'
- minus '-'
- address operator '@'
- star '*'
- type
- label
- object (parameter, function, local variable, global variable)

Example

```
asm lda ;  // error
```

26.1.501 C18104: `}' expected

[ERROR]

Description

This error occurs if an asm block (assembly code enclosed between braces) was not correctly marked as closed using }.

Example

```
asm { lda #1 //Error
```

26.1.502 C18105: Unexpected `@'

[ERROR]

Description

The "@" assembly operator can be used only with global variables. Applied in front of a variable it will refer to the address of that variable.

Example

```
asm  lda @100  /*Error: constant values aren't accepted
after @ operator*/


asm  lda @local_var /*Error: local values aren't accepted
after @ operator*/
```

26.1.503 C18107: Illegal operands

[ERROR]

Description

The inline assembly instruction operands do not match any of the instruction's addressing modes.

Example

```
asm JSR X



/*Error: the addressing mode with one parameter (which
requires an immediate operand) is used but instead of an
integer value a register is passed.*/
```

26.1.504 C18108: Address expected

[ERROR]

Description

This error message is issued if the assembler is set on to reject assembly instructions with indexed addressing mode.

Example

```
asm LDA ,X



//Error if indexed mode is not allowed
```

26.1.505 C18109: `!' expected

[ERROR]

Description

The HLI assembler expects an exclamation mark inline assembly operator.

Example

```
asm RTS  {}, {A} //Error: the exclamation mark was
omitted
```

26.1.506 C18110: Comma expected

[ERROR]

Description

A comma that delimits the operands of the assembly instruction was omitted.

Example

```
asm BRSET 0, 100 2 //Error: correct is asm BRSET 0, 100, 2
```

26.1.507 C18111: Constant expected

[ERROR]

Description

The assembly instruction used accepts only constant(s) as parameter(s). Local or global objects/variables cannot be used even if they are declared as const.

Example

```
int const local_var = 1.1


asm DCF local //Error. Correct is asm DCF 1.1
```

26.1.508 C18112: Bitno range expected

[ERROR

Description

One of the instruction's parameters is out of range.

Example

```
BRSET 9, 1, 2 //Error. First parameter of the BRSET
instruction can only takes values between 0 and 7.
```

Tips

Consult the architecture manual for the instruction operands to check if there are constraints related to accepted values.

26.1.509 C18113: Bitno expected

[ERROR]

Description

The HLI assembler expects a bit number.

Example

```
asm BSET i,j;  // correct is BSET 3,j
```

Tips

Do not use labels or objects for bit numbers.

26.1.510 C18114: Bitno expected

[ERROR

Description

The HLI assembler expects one of the following registers: A, H, X or HX.

26.1.511 C18115: `{` expected

[ERROR

Description

The left brace of the one of the register lists was omitted.

Example

```
asm RTS ! {}, A} //Error. Correct is


asm RTS ! {}, {A}
```

26.1.512 C18116: `}' or register expected

[ERROR

Description

The right brace of one of the register lists was omitted, or an incorrect register has been passed.

Example

```
asm RTS ! {    , A} //Error


asm RTS ! {Z}, A} //Error
```

26.1.513 C18117: Immedsiate/ Global address expected

[ERROR

Description

The immediate addressing mode can be used only with global variables or integer values. This error is issued if #label, #register or #local_var are used.

Example

```
asm LDA #local_var //Error
```

26.1.514 C18118: Label expected

[ERROR

Description

The HLI assembler expects a label or global variable operand.

Example

```
asm{

        label:

        ......

        JMP label:1 ; C18118: the inline assembler expects
a label (not an offset to a label)

}
```

26.1.515 C18119: Illegal frame specifier

[ERROR

Description

Illegal use of the ":" operator.

Example

```
label:


    LDA label:MSB
```

26.1.516 C18120: :Operator not allowed

[ERROR

Description

This message occurs if an assembly operator is misused.

Example

```
__asm BRSET 1, w:PAGE, label
```

The PAGE operator is not allowed in this context, hence C18120 occurs.

26.1.517 C18121: Object offset for X allowed only

[ERROR

Description

The message occurs for code such as: LDA @var,SP. Thus, the only register allowed here is X. For example: LDA @var, X. If any other register name (A, H, SP) else is used, then you get an error message.

26.1.518 C18122: Immediate or label expected

[ERROR

Description

The HLI assembler expects a label/global variable or an immediate operand.

Example

```
void foo{}


asm BSR foo //correct


asm BSR @foo //incorrect
```

26.1.519 C18123: end of the line expected

[ERROR

Description

The HLI assembler issues this error message when the assembler block or the assembly instruction has incorrect syntax.

Example

```
asm LDA #1} //Error
```

26.1.520 C18124: Immediate expected

[ERROR

Description

The HLI assembler expects an immediate operand.

Example

```
asm AIS local //Error: AIS instruction accepts as
operands only constant values
```

26.1.521 C18125: Invalid opcode or `:' expected

[ERROR

Description

The HLI assembler expects an instruction mnemonic or a label definition.

Tips

Check if the instruction/directive mnemonic is correctly typed or if the label definition is followed by a colon character.

26.1.522 C18126: Symbol redefined

[ERROR

Description

Two or more labels have been defined using the same name.

Example

```
asm{
```

```
    label:

    .....

    JMP label

    .....

    label:

}
```

26.1.523 C18127: Label, instruction, or directive expected

[ERROR

Description

The HLI assembler expects an instruction mnemonic, a label definition or a directive.

Example

```
asm LD //Error: this instruction mnemonic is not part of
the HCS08 instruction set.
```

Tips

Check if the instruction/directive mnemonic is correctly typed or if the label definition is followed by a colon character.

26.1.524 C18602: Displacement too large

[ERROR

Description

The inline assembly instruction uses a jump address that is too large: the value of the address cannot fit in the bits reserved for this purpose in the instruction machine code.

Example

```
asm BRSET 1, 2, 129 // Error: available addresses for
BRSET are in range [-128,127].
```

26.1.525 C18700: Unknown Opcode Operand Combination: Opc.:<Instr>/Dest.:<mode>/Source:<mode>.

[ERROR]

Description

There is an illegal combination of HC08 instruction mnemonic and src/dst operands. This is normally the case if using HLI an illegal addressing mode is used

Tips

Check the addressing modes if they are illegal, e.g some instructions work only with direct addressing mode.

26.1.526 C18701: Unknown Opcode

[ERROR

Description

The mnemonic is not recognised as a valid one.

Tip

Check if the instruction mnemonic was correctly typed.

26.1.527 C18702: Bitfield width exceeds 16

[ERROR

Description

The bitfield's size is greater than the size of the base type.

Example

```
typedef struct bf


{


        unsigned int : 22;


            //Error: the bit field size should be lower
or  equal than 16


}bf;
```

26.1.528 C20000: Dead code detected.

[WARNING]

Description

The low level debug information generator detected unreachable code, that has not been optimized by higher level parts of the compiler.

Tips

Check the C or HLI sources if they are really necessary.

26.1.529 C20001: Different value of stackpointer depending on control-flow.

[WARNING]

Description

The values of the stack pointer do not match at a joining point of the code. The reason for this serious warning may be

- inline assembly code the stack pointer cannot be traced
- the compiler may generated wrong code

Tips

Check your sources for a HLI code sequence from which the stack pointer value cannot be calculated at compile time. Check the listing file for correctness of the compiler. Please report bugs to your support. If you do not use HLI, map this warning to an error.

26.1.530 C20062: Ignored directive

[WARNING]

Description

This directive is accepted but not handled by the inline assembler.

26.1.531 C20085: Not a valid constant

[ERROR

Description

The operand is not recognized as a valid value for the assembly instruction.

Example

```
BRA -1 //Error: -1 is not accepted as value for the branch
instructions.
```

26.1.532 C20099: Cannot take address difference between local varaiables

[ERROR

Description

The inline assembler operator "-" can not be used with operators of address type.

Example

```
asm LDA (@local1 - @local2)      //Error
```

26.1.533 C20100: Out of spill locations: Reduce ? - expression

[ERROR]

Description

This error is issued for code having extremely many temporary locations active at one point. This limit is not reached by normal code.

Tips

Recode this part of the application. Use less ? operators.

26.1.534 C20110: Danger: access below stack pointer.

[WARNING]

Description

The low level debug information generator detected a stack pointer relative indirect memory access with an offset <= 0. Accesses below the stack pointer do not make any sense. This message may also come from HLI code, from which the stack pointer cannot be properly calculated at compile time, or from wrong code generated by the compiler.

Tips

Check your sources for a HLI code sequence from which the stack pointer value cannot be calculated at compile time. Check the listing file for correctness of the compiler. Please report bugs to your support. If you do not use HLI, map this warning to an error.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWMCUS08CMPREF
Revision 10.6, 01/2014