CodeWarrior Development Studio for Microcontrollers V10.x RS08 Build Tools Reference Manual

Document Number: CWMCURS08CMPREF
Rev 10.6, 01/2014

Contents

Chapter 10 ANSI-C Frontend

Section number	Title	Page

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

Section number Title Page

Chapter 1
Overview

The RS08 Build Tools Reference Manual for Microcontrollers describes the ANSI-C/C++ Compiler used for the Freescale 8-bit MCU (Microcontroller Unit) chip series.

NOTE

The technical notes and application notes are available in this location, <*CWInstallDir*>`\MCU\Help\PDF`, *where CWInstallDir* is the directory in which the CodeWarrior software is installed.

This document contains these major 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, Troubleshooting, and Technical Notes

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 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 CodeWarrior Development Studio and the compiler.
- Graphical User Interface : Describes the compiler's GUI.
- Environment : Describes all 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 macros predefined by the compiler.
- Compiler Pragmas : Lists the available pragmas.
- ANSI-C Frontend : Describes the ANSI-C implementation.
- Generating Compact Code : Describes the programming guidelines for the developer to produce compact and efficient code.
- RS08 Backend : Describes the code generator and basic type implementation, also information about hardware-oriented programming (optimizations, interrupt, and functions) specific for RS08.
- High-Level Inline Assembler for the Freescale RS08 : Describes the HLI Assembler for the RS08 compiler.

Chapter 3 Introduction

This chapter describes the RS08 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 RS08. Chapters one and two describe the configuration and creation of projects that target RS08 microcontrollers. Subsequent chapters describe the Compiler in greater detail.

This chapter consists of the following topics:

- Compiler Environment
- Creating and Managing Project Using CodeWarrior IDE
- Using Standalone Compiler
- Build Tools (Application Programs)
- Startup Command-Line Options
- Highlights
- CodeWarrior Integration of Build Tools
- Integration into Microsoft Visual C++ 2008 Express Edition (Version 9.0 or later)
- Object-File Formats

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 Build Tool utilities. The Build Tool utilities include the Linker, Assembler, ROM Burner, Simulator, and Debugger.

In general, a compiler t ranslates 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 generate 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 the assembly and C source code in a single project, refer to the High-Level Inline Assembler for the Freescale RS08 chapter.

The typical configuration of the Compiler is its association with a Project Directory and an Editor. A 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.

3.1.1 Designing a Project

There are three methods of designing a project.

- Use CodeWarrior IDE to coordinate and manage the entire project (Creating and Managing Project Using CodeWarrior IDE),
- Begin project construction with CodeWarrior IDE and use the standalone build tools to complete the project (Using Standalone Compiler),
- 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 this folder is: `C:\Freescale\CW MCU V10.x\MCU\prog`

3.2 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.

3.2.1 Creating Project Using New Bareboard 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**.

   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 Microcontrollers V10.x launches.

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. Specify a name for the new project. For example, enter the project name as `Project1`.
5. Click **Next**.

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

6. Select the desired CPU derivative for the project. For this example, select **RS08> RS08KA Family> MC9RS08KA1**.

   NOTE

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

7. Click **Next**.

   The **Connections** page appears.

8. 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.
9. Click **Next**.

The **Languages** page appears.

10. From the **Languages** options, check **C**. (This is the default setting).

NOTE

To enable the **Absolute Assembly** checkbox, clear the **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.

11. Click **Next**.

    The **Rapid Application Development** page appears.

12. Select the appropriate rapid application development tool.
13. Click **Next**.

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

14. Select **Small** memory model for memory model, **None**for the floating-point numbers format, and **ANSI startup code** for the level of startup code.

NOTE

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

NOTE

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

15. Click **Finish**.

The `Project1` project appears in the **CodeWarrior Projects** view in the Workbench window.

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).

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 `Project1` 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**.

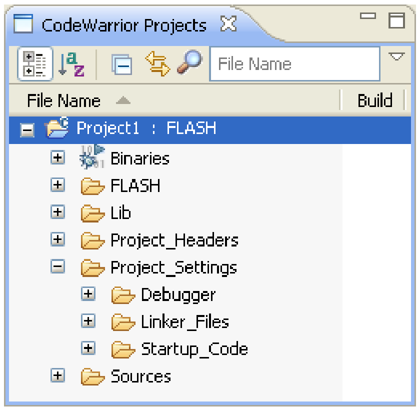


Figure 3-1. *CodeWarrior Projects View - Displaying Project1*

NOTE

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

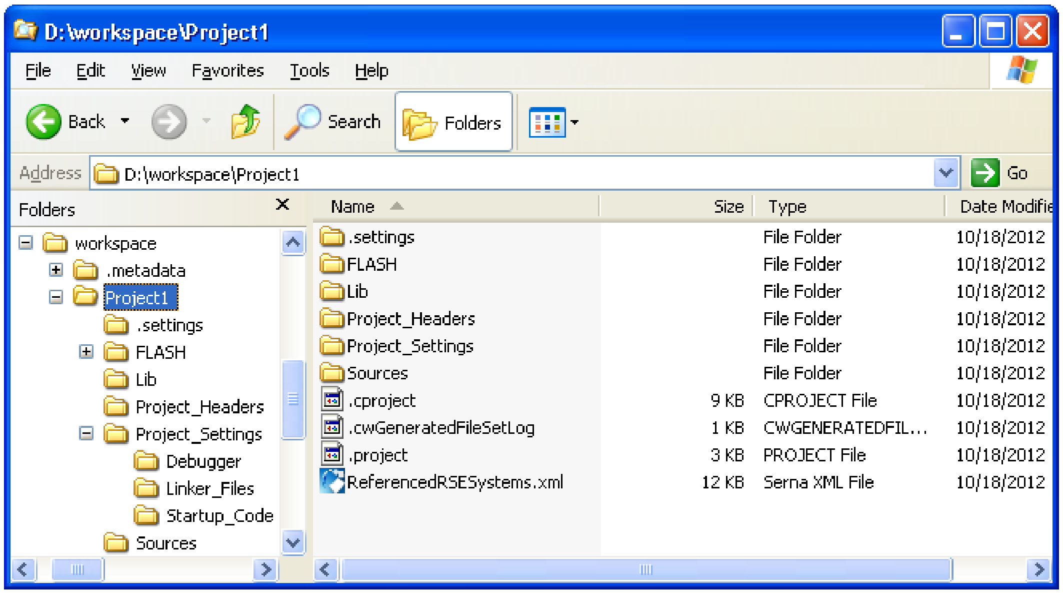
The view displays the logical arrangement of the files in the `Project1` project directory. At this stage, you can safely close the project and reopen it later, if desired.

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 `Project1`. This includes the source, header, 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 MCU derivative's registers and the symbols used to access them.

- `Project_Headers` is the directory that contains any Microcontrollers-specific header files.
- `Project_Settings` group contains the `Debugger` folder, `Linker_Files` folder and the `Startup_Code` folder. The `Debugger` folder contains any initialization and memory configuration files that prepare the hardware target for debugging. It also stores the launch configuration used for the debugging session. 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.
- `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 on the project's name (`Project1 : FLASH`) in the **CodeWarriorProjects** view, and select **Show In Windows Explorer** . Windows displays the Eclipse workspace folder, along with the project folder, `Project1`, within it, as the following image shows:



Name	Size	Type	Date Modifie
.settings		File Folder	10/18/2012
FLASH		File Folder	10/18/2012
Lib		File Folder	10/18/2012
Project_Headers		File Folder	10/18/2012
Project_Settings		File Folder	10/18/2012
Sources		File Folder	10/18/2012
.cproject	9 KB	CPROJECT File	10/18/2012
.cwGeneratedFileSetLog	1 KB	CWGENERATEDFIL...	10/18/2012
.project	3 KB	PROJECT File	10/18/2012
ReferencedRSESystems.xml	12 KB	Serna XML File	10/18/2012

Figure 3-2. Contents of Project1 Directory

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

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

3.2.2 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 `Project1` folder.

The default groups and their usual functions are:

- `FLASH`

  The `FLASH` 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 chosen Microcontrollers derivative. For this example, the `MC9RS08KA1.c` file supports the `MC9RS08KA1` 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, plus 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 MCU's initialization and startup functions. For RS08 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 on the source files and select **Exclude from Build** to prevent the build tools from compiling them.

3.2.3 Analysis of Files in CodeWarrior Projects View

Expand the groups by clicking your mouse 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 c ode 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`
- `mc9rs08ka1.c`

  located in <project_directory>\ `Lib`

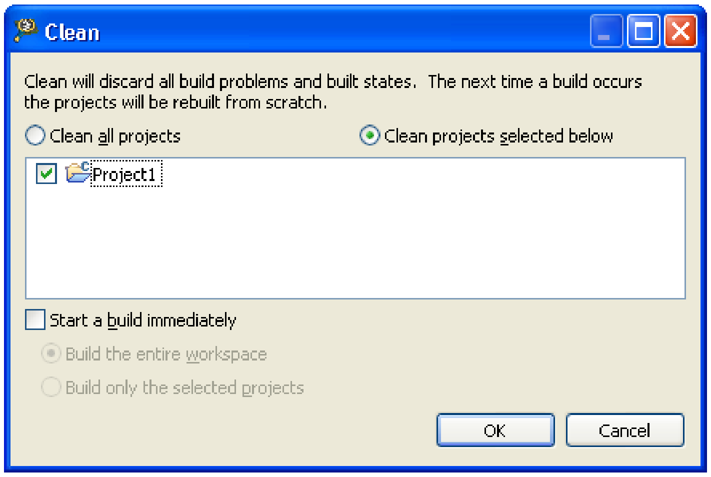
At this time, the project should be configured correctly and the source code should be free of syntactical errors. If the project has been built earlier, you should see a link to the project's binary files, and the `FLASH` folder present in the CodeWarrior Projects view.

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.

2. Select the **Clean projects selected below** option and check the project you want to build again.
3. Clear the **Start a build immediately** checkbox, if you want to build the project manually after cleaning.



Clean will discard all build problems and built states. The next time a build occurs the projects will be rebuilt from scratch.

Clean all projects [✓ Clean projects selected below]

[✓] Project1

[ ] Start a build immediately

[✓ Build the entire workspace]

Build only the selected projects

Figure 3-3. Clean Dialog Box

4. Click **OK**.

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

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

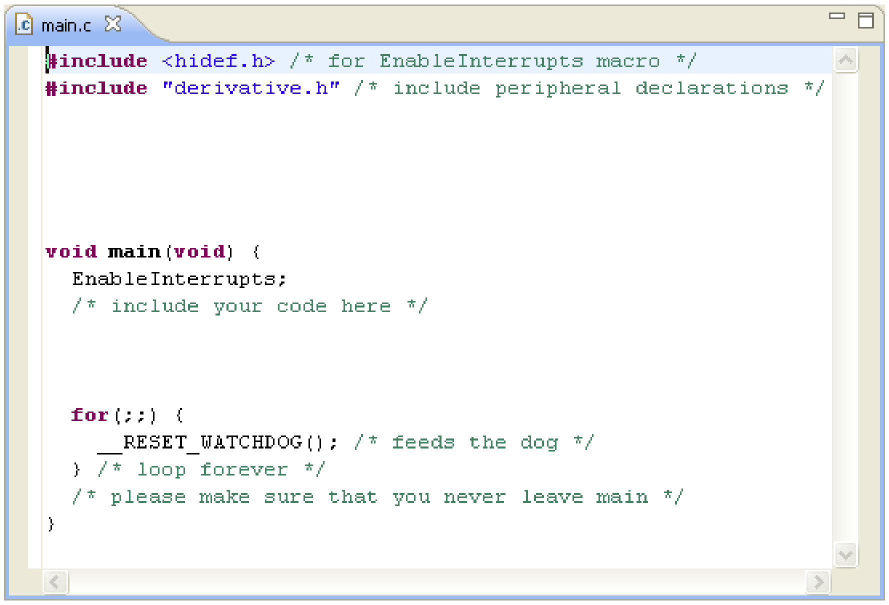
The **Console** view displays the statements that direct the build tools to compile and link the project. The `Binaries` link appears, and so does the `FLASH` folder.

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 Project1.abs` file.

The Linker Map file, `Project1.map`, file indicates the memory areas allocated for the program and contains other useful information.

To examine the source file, `main.c`, double click on the `main.c` file in the `Sources` group.

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


main.c

```
#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */





void main(void) {
  EnableInterrupts;
  /* include your code here */



  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Figure 3-4. 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.

The Graphical User Interface chapter provides information about configuring the options for the Compiler and other Build Tools, as well as information about feedback messages.

3.3 Using Standalone Compiler

You can use the RS08 compiler as a standalone compiler. This tutorial does not create an entire project with the Build Tools, but instead uses parts of a project already created by the CodeWarrior **New Bareboard Project** *wizard*.

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.

3.3.1 Configuring Compiler

Build tools use a tool directory for configuring and locating its generated files. For 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.

A build tool such as the Compiler requires information from 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 lists the typical locations for a global configuration file.

  Listing: Typical Locations for a 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`

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

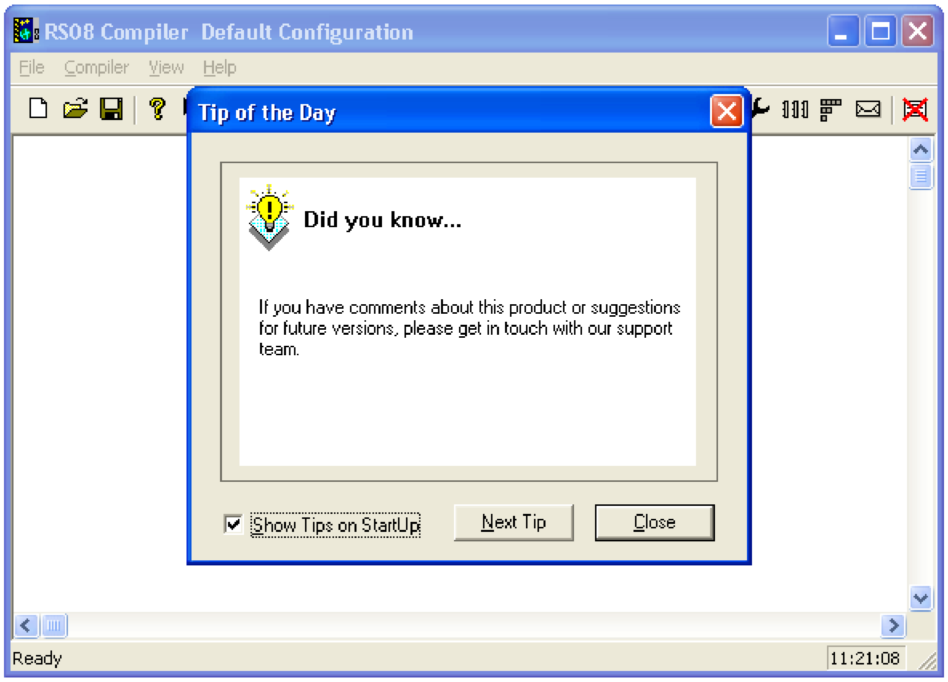
  For information about entries for the global configuration file, refer to the Global Configuration File Entries topic in the Appendices.

- 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 topic in the Appendices.

To configure the Compiler, proceed as follows:

1. Double-click `crs08.exe`, which is located in the <`CWInstallDir`>\\`MCU`\\`prog` folder.
2. The **RS08 Compiler** window appears.


RS08 Compiler Default Configuration

Tip of the Day

Did you know...

If you have comments about this product or suggestions for future versions, please get in touch with our support team.

Figure 3-5. RS08 Compiler Window

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

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

4. Select **File > New / Default Configuration** from the **RS08 Compiler** window menu bar.

NOTE

You can also use an existing configuration file if you desire. Use **File > Load Configuration** to open and read the configuration file.

5. Select **File > Save Configuration** (*or* **Save Configuration As**).

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

6. Browse and select the folder where you want to save the configuration. In this example, folder `x15` is used to save the configuration.

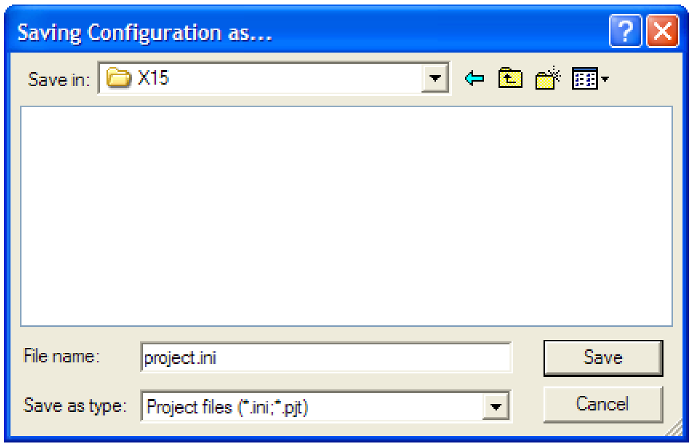
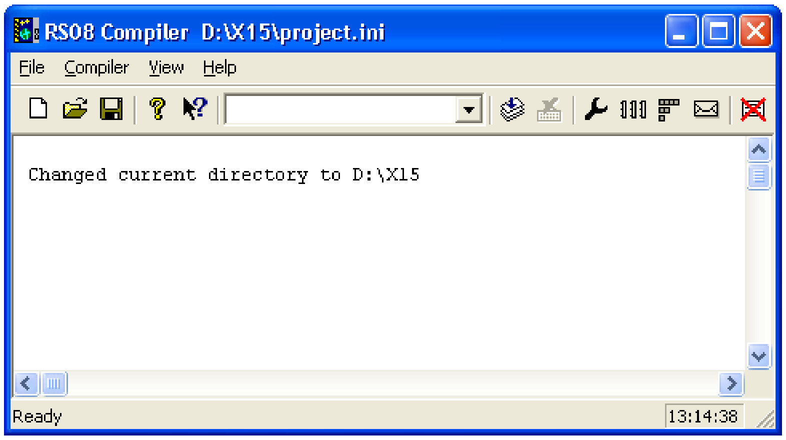

Saving Configuration as...
Save in: X15
File name: project.ini
Save as type: Project files (*.ini;*.pjt)
Save
Cancel

Figure 3-6. Saving Configuration as Dialog Box

7. Click **Save**.

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



```
Changed current directory to D:\X15
```

Figure 3-7. Compiler's Current Directory Switches to your Project Directory

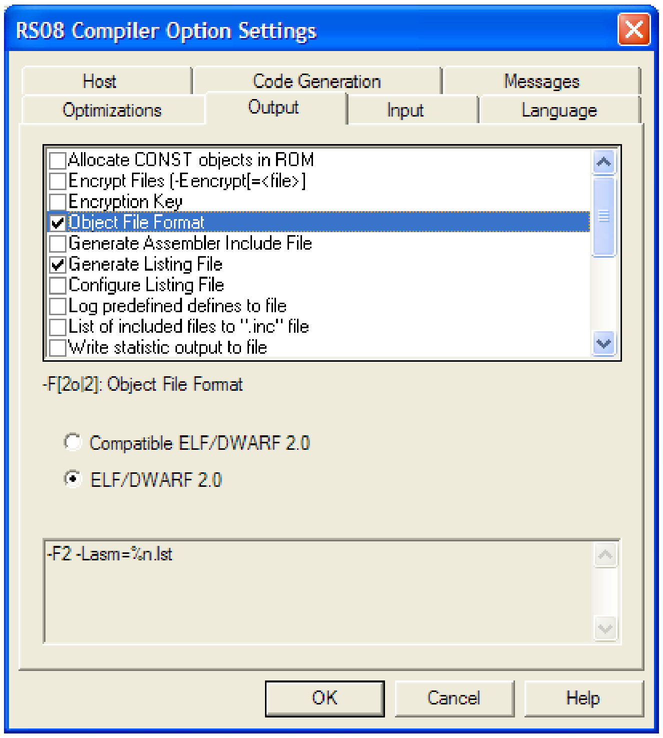
   The project directory folder now contains the newly-created `project.ini` configuration file. The project configuration file contains a copy of the `[CRS08_Compiler]` portion of the `project.ini` file in the build tools `prog` folder. If a build tool modifies any project option, the changes are specified 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.

8. Select **Compiler > Options > Options** from the **RS08 Compiler** window menu bar.

   The compiler displays the **RS08 Compiler Option Settings** dialog box.


RS08 Compiler Option Settings

- [ ] Allocate CONST objects in ROM
- [ ] Encrypt Files (-Eencrypt[=<file>]
- [ ] Encryption Key
- [✓] Object File Format
- [ ] Generate Assembler Include File
- [✓] Generate Listing File
- [ ] Configure Listing File
- [ ] Log predefined defines to file
- [ ] List of included files to ".inc" file
- [ ] Write statistic output to file

-F[2o|2]: Object File Format

Compatible ELF/DWARF 2.0 [✓ ELF/DWARF 2.0]

```
-F2 -Lasm=%n.lst
```

Figure 3-8. RS08 Compiler Option Settings Dialog Box

9. Select the **Output** tab.
10. Check the **Generate Listing File**checkbox.
11. Check the **Object File Format** checkbox and select the **ELF/DWARF 2.0** option which appears for the **Object File Format** checkbox.
12. Click **OK**.

    The **RS08 Compiler Option Settings** dialog box closes.

13. Save the changes to the configuration by either:
    - selecting **File > Save Configuration** *(*`Ctrl + s`*)* or
    - clicking **Save** on the toolbar.

The compiler is now configured with the selected settings.

NOTE

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

3.3.1.1 Selecting Input Files

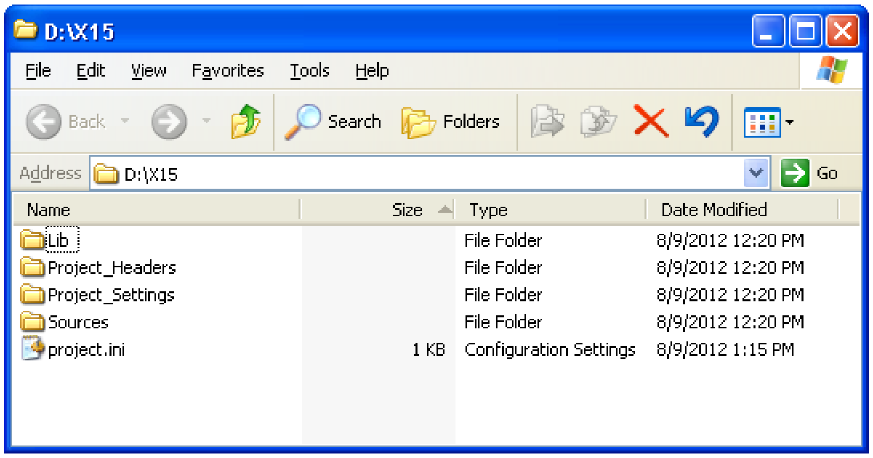
The project does not contain any source code files at this point. Copy and paste the C source code (`*.c`) and header (`*.h`) files from the previous project, `Project1`, to save time.

1. Copy the `Sources` folder from the `Project1` folder and paste it into the `X15` project folder.
2. Copy the `Project_Headers` folder from the `Project1` folder and paste it into the `X15` project folder.
3. Copy and paste the `Project1` 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 `Project1\Project_Settings` folder into the `X15\Project_Settings` folder.

The `X15` project folder have the required C source code (`*.c)` and header (`*.h)` files from the previous project.

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 `MC9RS08KA1.h header file`
- the `Lib` folder, which contains the `MC9RS08KA1.c source file`
- the `Project_settings\Startup_Code` folder, which contains the start up and initialization file, `start08.c`.



Name	Size	Type	Date Modified
Lib		File Folder	8/9/2012 12:20 PM
Project_Headers		File Folder	8/9/2012 12:20 PM
Project_Settings		File Folder	8/9/2012 12:20 PM
Sources		File Folder	8/9/2012 12:20 PM
project.ini	1 KB	Configuration Settings	8/9/2012 1:15 PM

Figure 3-9. Folders and Files in X15 Project Folder

3.3.1.2 Compiling C Source Code Files

Now compile the `start08.c` file.

1. Select **File > Compile** from the **RS08 Compiler** window menu bar.

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

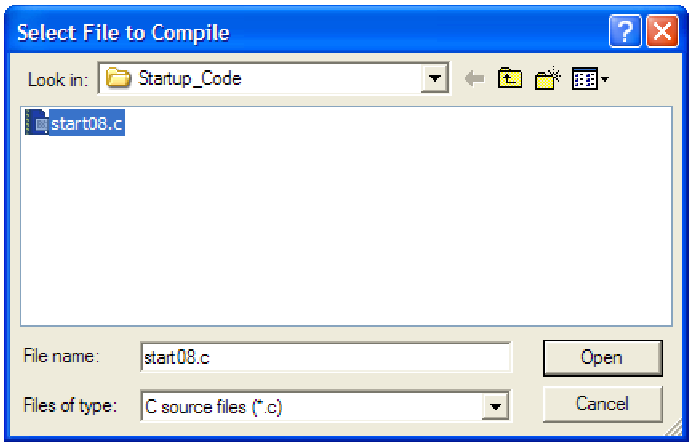
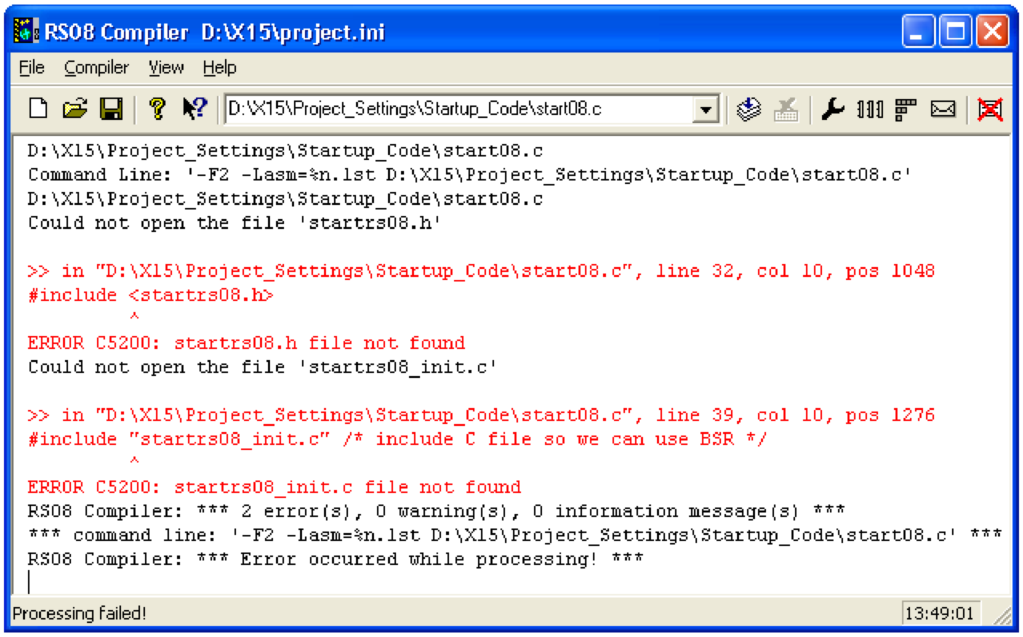


Figure 3-10. Select File to Compile Dialog Box

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. In this case, the file fails to compile.


RS08 Compiler D:\X15\project.ini
File Compiler View Help
D:\X15\Project_Settings\Startup_Code\start08.c
Processing failed!
13:49:01

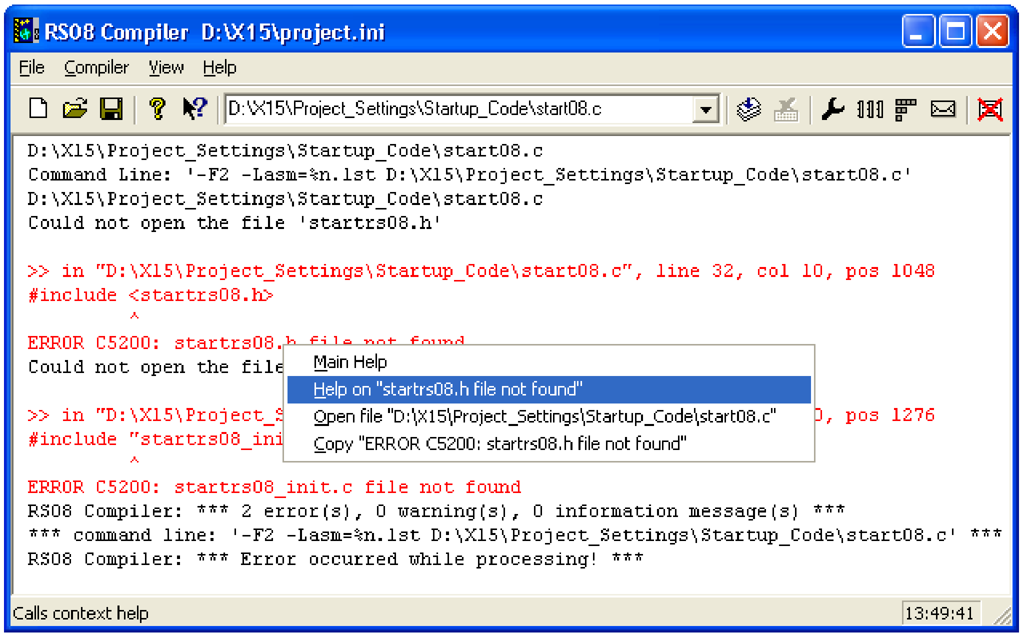
```
D:\X15\Project_Settings\Startup_Code\start08.c
Command Line: '-F2 -Lasm=%n.lst D:\X15\Project_Settings\Startup_Code\start08.c'
D:\X15\Project_Settings\Startup_Code\start08.c
Could not open the file 'startrs08.h'

>> in "D:\X15\Project_Settings\Startup_Code\start08.c", line 32, col 10, pos 1048
#include <startrs08.h>
         ^
ERROR C5200: startrs08.h file not found
Could not open the file 'startrs08_init.c'

>> in "D:\X15\Project_Settings\Startup_Code\start08.c", line 39, col 10, pos 1276
#include "startrs08_init.c" /* include C file so we can use BSR */
         ^
ERROR C5200: startrs08_init.c file not found
RS08 Compiler: *** 2 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst D:\X15\Project_Settings\Startup_Code\start08.c' ***
RS08 Compiler: *** Error occurred while processing! ***
```

Figure 3-11. Results of Compiling start08.c File

The RS08 Compiler window provides information about the compilation process and generates error messages if the compilation fails. In this case the `C5200: `FileName' file not found` error message appears twice, once for `startrs08.h` and once for `startrs08_init.c`.

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

A context menu appears.

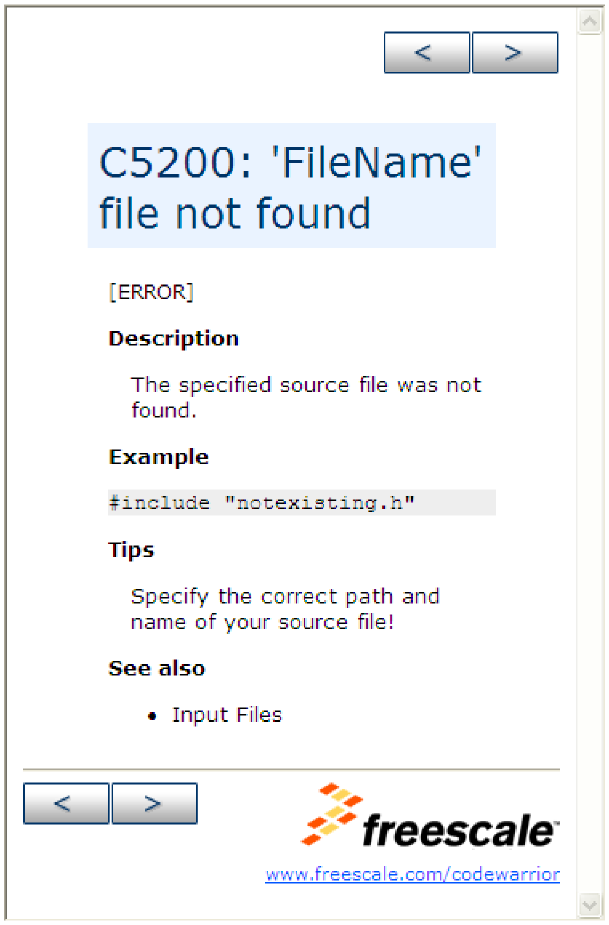

RS08 Compiler D:\X15\project.ini
File Compiler View Help
D:\X15\Project_Settings\Startup_Code\start08.c
Main Help
Help on "startrs08.h file not found"
Open file "D:\X15\Project_Settings\Startup_Code\start08.c"
Copy "ERROR C5200: startrs08.h file not found"
Calls context help
13:49:41

```
D:\X15\Project_Settings\Startup_Code\start08.c
Command Line: '-F2 -Lasm=%n.lst D:\X15\Project_Settings\Startup_Code\start08.c'
D:\X15\Project_Settings\Startup_Code\start08.c
Could not open the file 'startrs08.h'

>> in "D:\X15\Project_Settings\Startup_Code\start08.c", line 32, col 10, pos 1048
#include <startrs08.h>
         ^
ERROR C5200: startrs08.h file not found
Could not open the file
>> in "D:\X15\Project_S                                                  0, pos 1276
#include "startrs08_ini
         ^
ERROR C5200: startrs08_init.c file not found
RS08 Compiler: *** 2 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst D:\X15\Project_Settings\Startup_Code\start08.c' ***
RS08 Compiler: *** Error occurred while processing! ***
```

Figure 3-12. RS08 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.



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

Figure 3-13. C5200 Error Message Help

The **Tips** section in the help f *or the* **C5200 error** states that the correct paths and names for the source files must be specified.

The following folder contains the missing header file, `startrs08.h`:

```
<CWInstallDir>
\
MCU\lib\rs08c\include
```

NOTE

The `#include` preprocessor directive for this header file appears on line 29 of `start08.c`.

The following folder contains the missing `startrs08_init.c` file.

`CWInstallDir\MCU\lib\rs08c\src`

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

1. Select **File > Configuration** from the **RS08 Compiler** window menu bar..

The **Configuration** dialog box appears.

2. Select the **Environment** tab in the **Configuration** dialog box.
3. Select **Header File Path**.

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 (File Processing).

4. Click "..." button.

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

5. In the **Browse for Folder** dialog box, navigate to the missing `startrs08.h` file in the `<CWInstallDir>\MCU\lib\rs08c\include` folder.

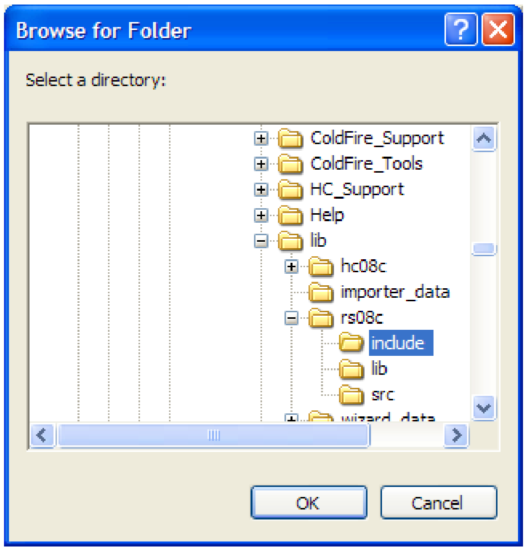


Figure 3-14. include Subfolder Containing startrs08.h

6. Click **OK**.

   The **Configuration** dialog box is now active.

7. Click the **Add** button.

   The specified path appears in the lower panel.

8. Click " **...** " button again.
9. In the **Browse for Folder** dialog, navigate to `\lib\hc08c\include` subfolder in the CodeWarrior installation folder.
10. Click **Add**.

The path to the header files `<CWInstallDir>\MCU\lib\hc08c\include` appears in the lower pane.

11. Select **General Path**.
12. Click " **...** " button.
13. In the **Browse for Folder** dialog, navigate to the missing `startrs08_init.c` file in the `rs08c\src` subfolder in the CodeWarrior installation's `lib` folder.
14. Click **Add**.

    The path to the header files < *CWInstallDir*>\`MCU\lib\rs08c\src` appears in the lower panel.

15. Click **OK**.

    An asterisk now appears in the **RS08 Compiler** window's title bar, indicating that the configuration file contains unsaved changes.

16. Click **Save** on the toolbar to save the configuration modifications. Alternatively, select **File > Save Configuration** from the **RS08 Compiler** window menu bar.

You have now saved the new paths to the missing files to compile the `start08.c` file.

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 supplied the paths to the missing files, you can try again to compile the `start08.c` file.

1. Select **File > Compile** from the **RS08 Compiler** window menu bar.
2. Navigate to the `Project_settings\Startup_Code` folder in the project directory.
3. Select `start08.c`.
4. Click **Open**.

The Compiler indicates successful compilation for the `start08.c` file.

The compiler displays following results:

- The `start08.c` file involved the use of six 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 117 bytes.
- The compiler created 11 global objects and the `Data Size` was 3 bytes in RAM, and 5 bytes in ROM.
- There were 0 errors, 0 warnings, and 0 information messages.

Tip

Clear the Compiler window at any time by selecting **View > Log > Clear Log**.

Now compile the `main.c` file:

1. Navigate to the `Sources` folder.
2. Select the `main.c` file.
3. 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 generated 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 **RS08 compiler** window menu bar.

   The **Configuration** dialog box appears.

2. Select the **Environment** tab.
3. Select **Header File Path**.
4. Click " **...** " button.

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

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

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

7. *Click* **Add**.

   The selected path appears the lower pane.

8. *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.*

9. *Select* **File > Compile** *from the* **RS08 Compiler** *window menu bar.*
10. *Select* `main.c` *in the* `Sources` *folder. You can also select a previously compiled C source code file.*
11. *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.

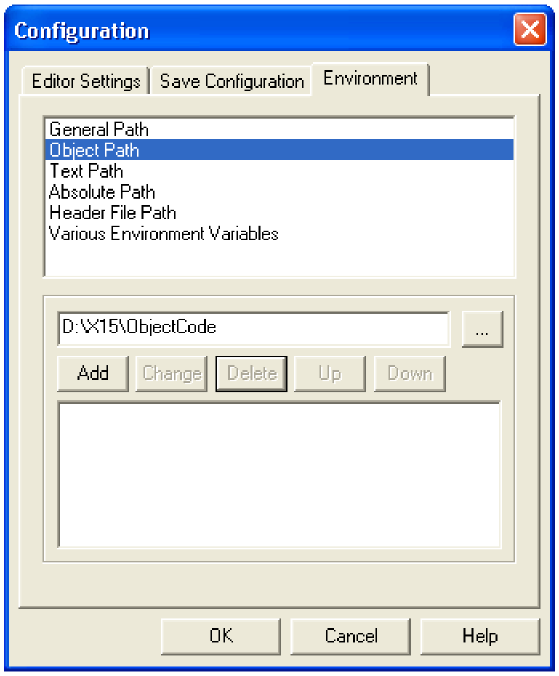
The compiler places the object file in the `Sources` folder along with the C source code file, and generates output listing files into the project folder. The binary object files and the input modules have identical names except for the extension used. Any assembly output files generated for each C source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

At this time, only two of the three C source code files have been compiled. Locate the remaining C source code file, `MC9RS08KA1.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 `MC9RS08KA1.c` file to another folder, modify the compiler configuration so that when the `MC9RS08KA1.c` 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**.
3. *Click the* **Environment** *tab and select* **Object Path**.
4. Navigate to the new object code folder, `ObjectCode`, in the project folder.
5. Click **OK**.


Configuration

Editor Settings | Save Configuration | Environment

General Path
Object Path
Text Path
Absolute Path
Header File Path
Various Environment Variables

D:\X15\ObjectCode

Figure 3-15. Adding OBJPATH

6. Click **Add.**

The new object path appears in the lower panel.

7. Click **OK**.

   The **Configuration** dialog box closes.

8. Press **Ctrl + S** to save the settings.
9. Select **File > Compile** from the **RS08 Compiler** window menu bar.

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

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

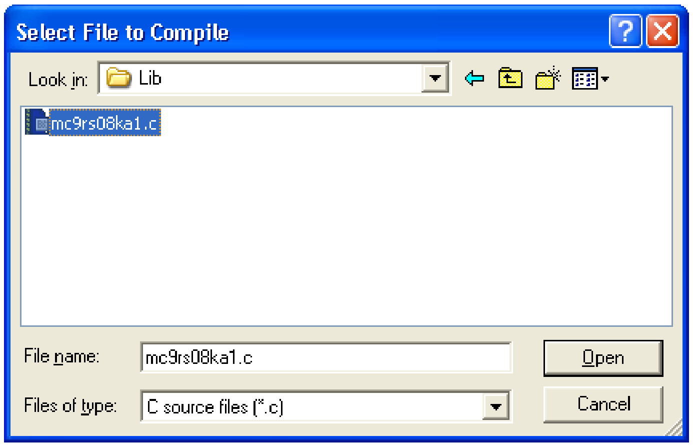


Figure 3-16. Compiling Source File in Lib Folder

11. Click **Open**.

The selected file compiles.

The compiler log states that `MC9RS08KA1.o`, the object code file, was created in the `ObjectCode` folder, as specified. Save the *configuration* again in case you wish to recompile any of the C source code files in the 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 properly configured 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 contains many derivative-specific header and other files available for inclusion into your 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.

For more information, refer to the *Linker* section in the *Build Tool UtilitiesManual*. However, the project set up and configuration is faster and easier using the **New Bareboard Project** wizard.

3.4 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
ahc08.exe	Freescale HC(S)08/RS08 assembler
burner.exe	Batch and interactive burner application that generates S-records
crs08.exe	Freescale RS08 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.

3.5 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 (as show in the listing below). For more information, refer to the -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 Slider** dialog box.
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 a Startup 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 the Current Project Settings File

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

3.6 Highlights

- 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

3.7 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 (usually `C:\Freescale\CW MCU V10.x\eclipse`). The plug-ins are installed in the `eclipse\plugins` directory.

3.7.1 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` (usually `C:\Freescale\CW MCU V10.x\MCU\bin`) directory. To do this, start the `regservers.bat` batch file.

3.7.2 RS08 Compiler Build Settings Panels

The following sections describe the settings panels that configure the RS08 Compiler build properties. 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, click on 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 table lists the build properties panels for the RS08 Compiler.

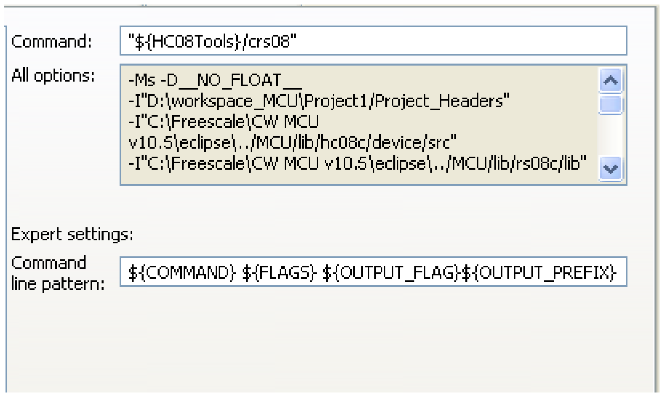
Table 3-3. Build Properties for RS08

Build Tool	Build Properties Panels
RS08 Compiler	RS08 Compiler > Output RS08 Compiler > Output > Configure Listing File RS08 Compiler > Output > Configuration for list of included files in make format
	RS08 Compiler > Input
	RS08 Compiler > Language RS08 Compiler > Language > CompactC++ features
	RS08 Compiler > Host
	RS08 Compiler > Code Generation
	RS08 Compiler > MessagesRS08 Compiler > Messages > Disable user messages
	RS08 Compiler > Preprocessor
	RS08 Compiler > Type Sizes
	RS08 Compiler > General
	RS08 Compiler > Optimization RS08 Compiler > Optimization > Mid level optimizations RS08 Compiler > Optimization > Mid level branch optimizations RS08 Compiler > Optimization > Tree optimizer RS08 Compiler > Optimization > Optimize Library Function

3.7.2.1 RS08 Compiler

Use this panel to specify the command, options, and expert settings for the build tool compiler. Additionally, the RS08 Compiler tree control includes the general, include file search path settings.

The following figure shows the RS08 Compiler settings.


Command: "${HC08Tools}/crs08"

All options:

```
-Ms -D__NO_FLOAT__
-I"D:\workspace_MCU\Project1/Project_Headers"
-I"C:\Freescale\CW MCU
v10.5\eclipse\../MCU/lib/hc08c/device/src"
-I"C:\Freescale\CW MCU v10.5\eclipse\../MCU/lib/rs08c/lib"
```

Expert settings:

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG}${OUTPUT_PREFIX}

Figure 3-17. Tool Settings - RS08 Compiler

The following table lists and describes the compiler options for RS08.

Table 3-4. Tool Settings - RS08 Compiler Options

Option	Description
Command	Shows the location of the compiler executable file. Default value is : " `${HC08Tools}/crs08.exe"`. You can specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the compiler will be called with.
Expert Settings	Shows the expert settings command line parameters; default is `{COMMAND} ${FLAGS}${OUTPUT_FLAG}${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.
Command line pattern	

3.7.2.2 RS08 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.

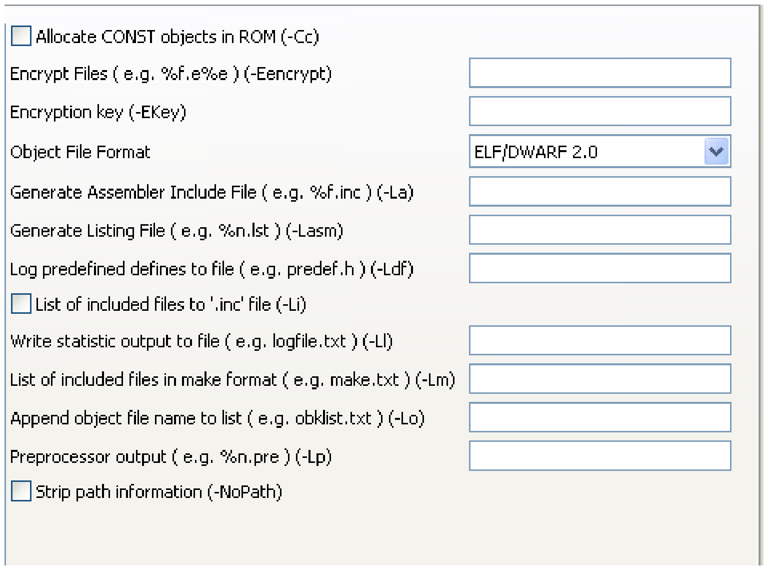

☐ Allocate CONST objects in ROM (-Cc)
Encrypt Files (e.g. %f.e%e) (-Eencrypt)
Encryption key (-EKey)
Object File Format: ELF/DWARF 2.0
Generate Assembler Include File (e.g. %f.inc) (-La)
Generate Listing File (e.g. %n.lst) (-Lasm)
Log predefined defines to file (e.g. predef.h) (-Ldf)
☐ List of included files to '.inc' file (-Li)
Write statistic output to file (e.g. logfile.txt) (-Ll)
List of included files in make format (e.g. make.txt) (-Lm)
Append object file name to list (e.g. obklist.txt) (-Lo)
Preprocessor output (e.g. %n.pre) (-Lp)
☐ Strip path information (-NoPath)

Figure 3-18. Tool settings - RS08 Compiler > Output

The following table lists and describes the output options for RS08 compiler.

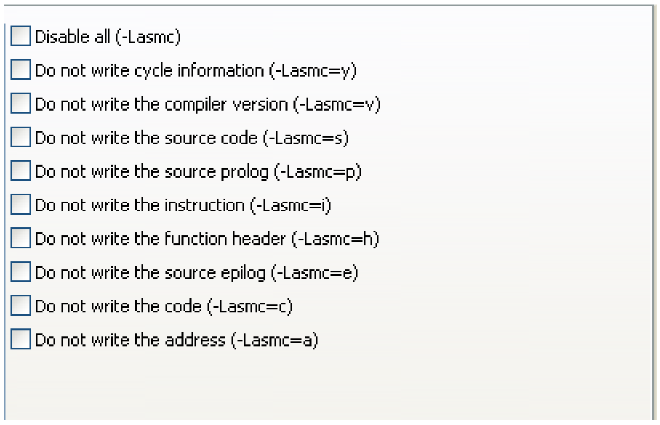
Table 3-5. Tool Settings - RS08 Compiler > Output Options

Option	Description
Allocate `CONST` objects in ROM (`-Cc`)	Refer to the -Cc: Allocate Const Objects into ROM topic.
Encrypt FIle (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 (-F2, -F2o): 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.
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.

3.7.2.3 RS08 Compiler > Output > Configure Listing File

Use this panel to configure the listing files for the RS08 Compiler to generate output.

The following figure shows **Output > ConfigureListing File** panel.



- [ ] Disable all (-Lasmc)
- [ ] Do not write cycle information (-Lasmc=y)
- [ ] Do not write the compiler version (-Lasmc=v)
- [ ] Do not write the source code (-Lasmc=s)
- [ ] Do not write the source prolog (-Lasmc=p)
- [ ] Do not write the instruction (-Lasmc=i)
- [ ] Do not write the function header (-Lasmc=h)
- [ ] Do not write the source epilog (-Lasmc=e)
- [ ] Do not write the code (-Lasmc=c)
- [ ] Do not write the address (-Lasmc=a)

Figure 3-19. Tool Settings - RS08 Compiler > Output > Configure Listing File

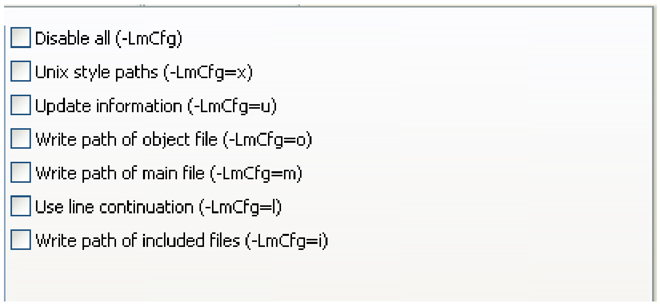
NOTE

For information about the description of the options available in the **Configure Listing File** panel for RS08 compiler, refer to the -Lasmc: Configure Listing File topic.

3.7.2.4 RS08 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 the RS08 Compiler to generate the output.

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



- [ ] Disable all (-LmCfg)
- [ ] Unix style paths (-LmCfg=x)
- [ ] Update information (-LmCfg=u)
- [ ] Write path of object file (-LmCfg=o)
- [ ] Write path of main file (-LmCfg=m)
- [ ] Use line continuation (-LmCfg=l)
- [ ] Write path of included files (-LmCfg=i)

Figure 3-20. Tool Settings ? RS08 Compiler > Output > Configuration for List of Included Files in Make Format

NOTE

For information about the description of the options available in the **Configuration for list of included files in make format** panel for RS08 compiler, refer to the -LmCfg: Configuration for List of Included Files in Make Format (option -Lm) topic.

3.7.2.5 RS08 Compiler > Input

Use this panel to specify file search paths and any additional include files the **RS08 Compiler** should use. You can specify multiple search paths and the order in which you want to perform the search.

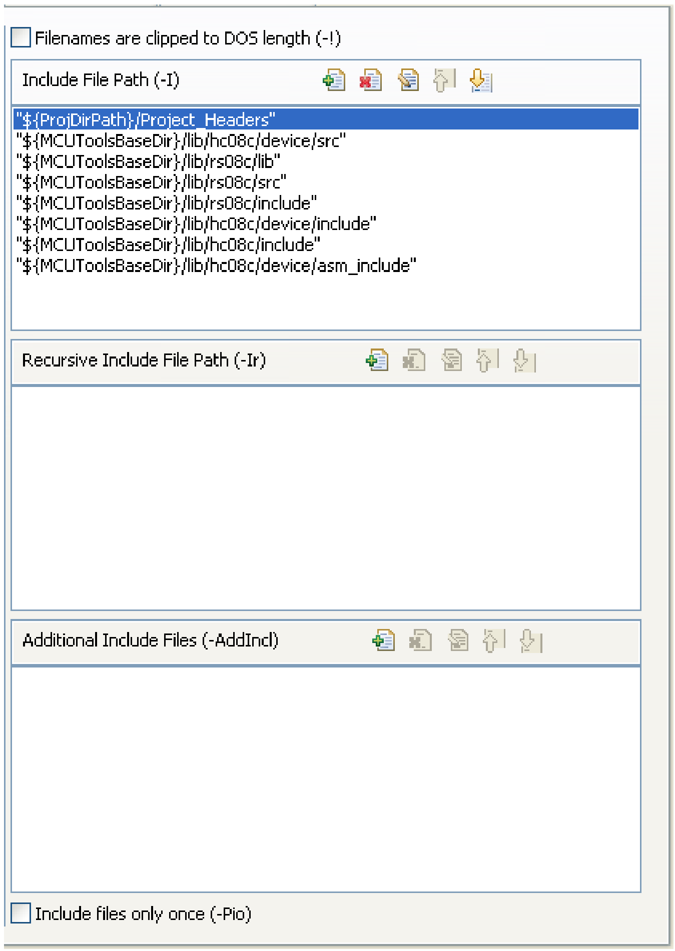
The IDE first looks for an include file in the current directory, or the directory that you specify in the `INCLUDE` directive. If the IDE does not find the file, it continues searching the paths shown in this panel. The IDE keeps searching paths until it finds the `#include` file or finishes searching 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.



Filenames are clipped to DOS length (-!)

Include File Path (-I)

```
"${ProjDirPath}/Project_Headers"
"${MCUToolsBaseDir}/lib/hc08c/device/src"
"${MCUToolsBaseDir}/lib/rs08c/lib"
"${MCUToolsBaseDir}/lib/rs08c/src"
"${MCUToolsBaseDir}/lib/rs08c/include"
"${MCUToolsBaseDir}/lib/hc08c/device/include"
"${MCUToolsBaseDir}/lib/hc08c/include"
"${MCUToolsBaseDir}/lib/hc08c/device/asm_include"
```

Recursive Include File Path (-Ir)

Additional Include Files (-AddIncl)

Include files only once (-Pio)

Figure 3-21. Tool Settings - RS08 Compiler > Input

The following table lists and describes the input options for RS08 Compiler.

Table 3-6. Tool Settings - RS08 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.
Recursive Include File Path (`-Ir`)	Use this option to add the required file paths recursively.
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. To confirm deletion, click **Yes** in the **Confirm Delete** dialog box.
	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.

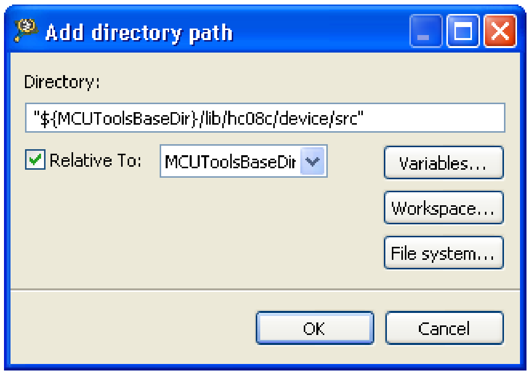


Figure 3-22. Tool Settings - RS08 Compiler > Input - Add directory path Dialog Box

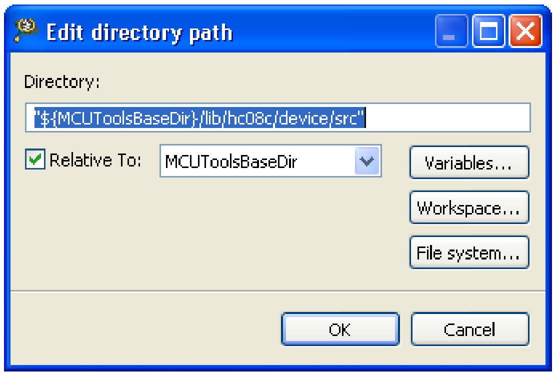


Figure 3-23. Tool Settings - RS08 Compiler > Input - Edit directory path Dialog Box

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.

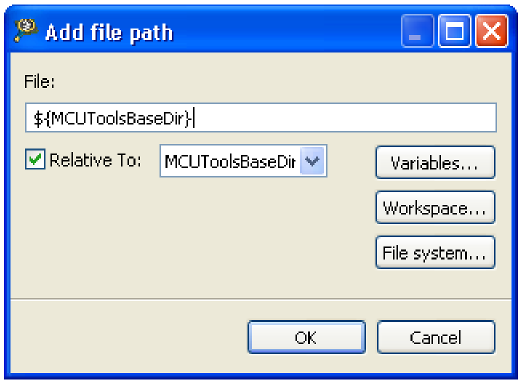
- **Workspace -** Click to display the **Folder Selection** dialog box and specify the path. The resulting path, relative to the workspace, appears in the appropriate list.
- **File system -** Click to display the **Browse For Folder** dialog box and specify the path. The resulting path appears in the appropriate list.
- **Variables -** Click to display the **Select build variable** dialog box. The resulting path appears in the appropriate list.
- **Relative To-** Check this checkbox to enable the build variables drop-down list. You can select the desired variable for the existing settings.

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 directory path** dialog box and specify location of the library you want to add.
	Delete - Click to delete the selected library path. To confirm deletion, click **Yes** in the **Confirm Delete** dialog box.
	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 file path** dialog box.


Add file path

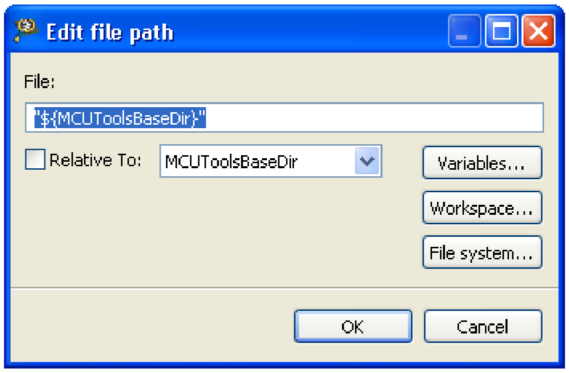
File:

${MCUToolsBaseDir}

☑ Relative To: MCUToolsBaseDir

Figure 3-24. Tool Settings - RS08 Compiler > Input - Add file path Dialog Box

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



"${MCUToolsBaseDir}"

MCUToolsBaseDir

Figure 3-25. Tool Settings - RS08 Compiler > Input - Edit file path Dialog Box

The buttons in the **Add file pathand 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 **Folder 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.
- **Relative To-** Check this checkbox to enable the build variables drop-down list. You can select the desired variable for the existing settings.

3.7.2.6 RS08 Compiler > Language

Use this panel to specify code-generation and symbol-generation options for the RS08 Compiler.

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

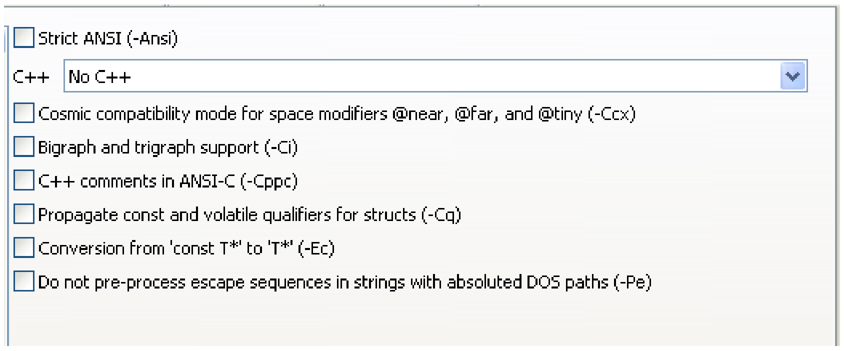

Strict ANSI (-Ansi)
C++ No C++
Cosmic compatibility mode for space modifiers @near, @far, and @tiny (-Ccx)
Bigraph and trigraph support (-Ci)
C++ comments in ANSI-C (-Cppc)
Propagate const and volatile qualifiers for structs (-Cq)
Conversion from 'const T*' to 'T*' (-Ec)
Do not pre-process escape sequences in strings with absoluted DOS paths (-Pe)

Figure 3-26. Tool Settings - RS08 Compiler > Language

The following table lists and describes the language options for RS08.

Table 3-9. Tool Settings - RS08 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.

3.7.2.7 RS08 Compiler > Language > CompactC++ features

Use the CompactC++ features panel to specify a compact C++ code for the RS08 Compiler.

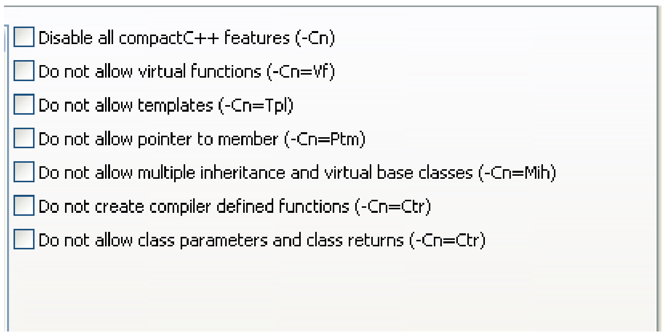

Disable all compactC++ features (-Cn)
Do not allow virtual functions (-Cn=Vf)
Do not allow templates (-Cn=Tpl)
Do not allow pointer to member (-Cn=Ptm)
Do not allow multiple inheritance and virtual base classes (-Cn=Mih)
Do not create compiler defined functions (-Cn=Ctr)
Do not allow class parameters and class returns (-Cn=Ctr)

Figure 3-27. Tool Settings - RS08 Compiler > Language > CompactC++ Features

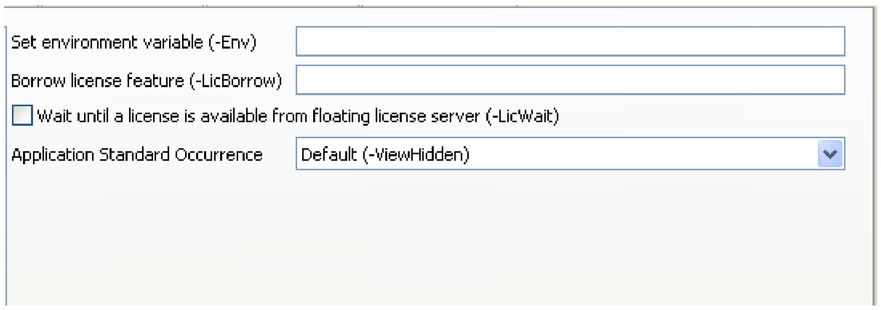
NOTE

For information about the description of the options available in the **CompactC++ features** panel for RS08 compiler, refer to the -Cn: Disable compactC++ features topic.

3.7.2.8 RS08 Compiler > Host

Use this panel to specify the host options of RS08.

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



Set environment variable (-Env)

Borrow license feature (-LicBorrow)

☐ Wait until a license is available from floating license server (-LicWait)

Application Standard Occurrence: Default (-ViewHidden)

Figure 3-28. Tool Settings - Host

The following table lists and describes the host options for RS08.

Table 3-10. Tool Settings - Host

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 License Server topic.
Application Standard Occurrence	Refer to the -View: Application Standard Occurrence topic.

3.7.2.9 RS08 Compiler > Code Generation

Use this panel to specify code- and symbol-generation options for the RS08 Compiler.

The following figure shows the **Code Generation** panel.

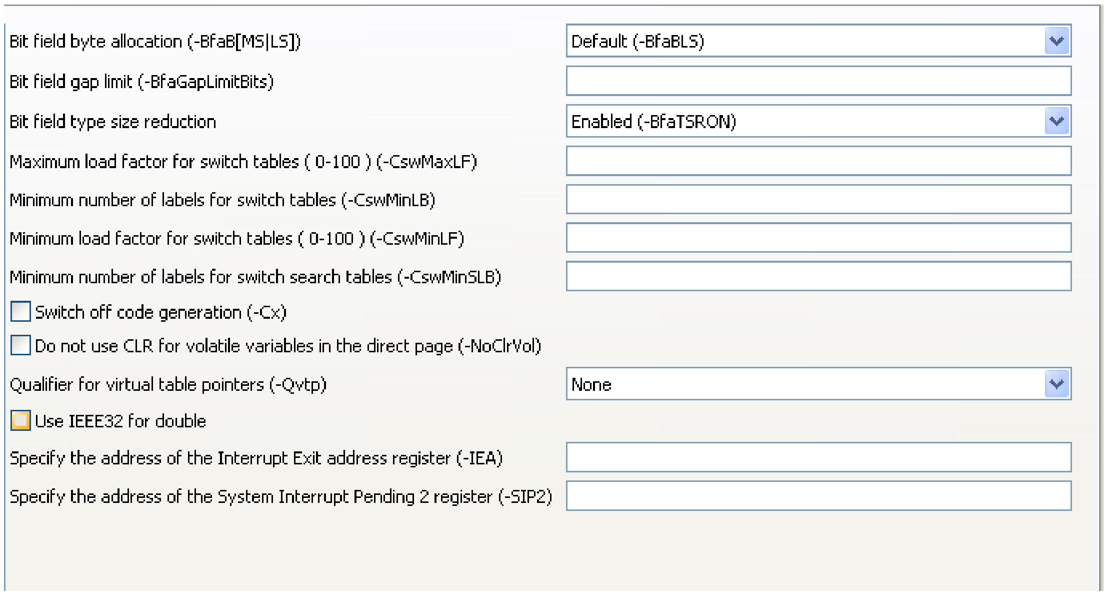

Bit field byte allocation (-BfaB[MS|LS]) Default (-BfaBLS)
Bit field gap limit (-BfaGapLimitBits)
Bit field type size reduction Enabled (-BfaTSRON)
Maximum load factor for switch tables (0-100) (-CswMaxLF)
Minimum number of labels for switch tables (-CswMinLB)
Minimum load factor for switch tables (0-100) (-CswMinLF)
Minimum number of labels for switch search tables (-CswMinSLB)
☐ Switch off code generation (-Cx)
☐ Do not use CLR for volatile variables in the direct page (-NoClrVol)
Qualifier for virtual table pointers (-Qvtp) None
☐ Use IEEE32 for double
Specify the address of the Interrupt Exit address register (-IEA)
Specify the address of the System Interrupt Pending 2 register (-SIP2)

Figure 3-29. Tool Settings - RS08 Compiler > Code Generation

The following table lists and describes the code generation options for RS08 compiler.

Table 3-11. Tool Settings - RS08 Compiler > Code Generation Options

Option	Description
Bit field byte allocation (`-BfaB[MS|LS]`)	Refer to the -BfaB: Bitfield Byte Allocation topic.
Bit field gap limits (`-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	Refer to the -Fd: Double is IEEE32 topic.
Specify the address of the Interrupt Exit address register (`-IEA`)	Refer to the -IEA: Specify the address of the interrupt exit address register topic.
Specify the address of the System Interrupt Pending 2 register (`-SIP2`)	Refer to the -SIP2: Specify the address of the System Interrupt Pending 2 register topic.

3.7.2.10 RS08 Compiler > Messages

Use this panel to specify whether to generate symbolic information for debugging the build target

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

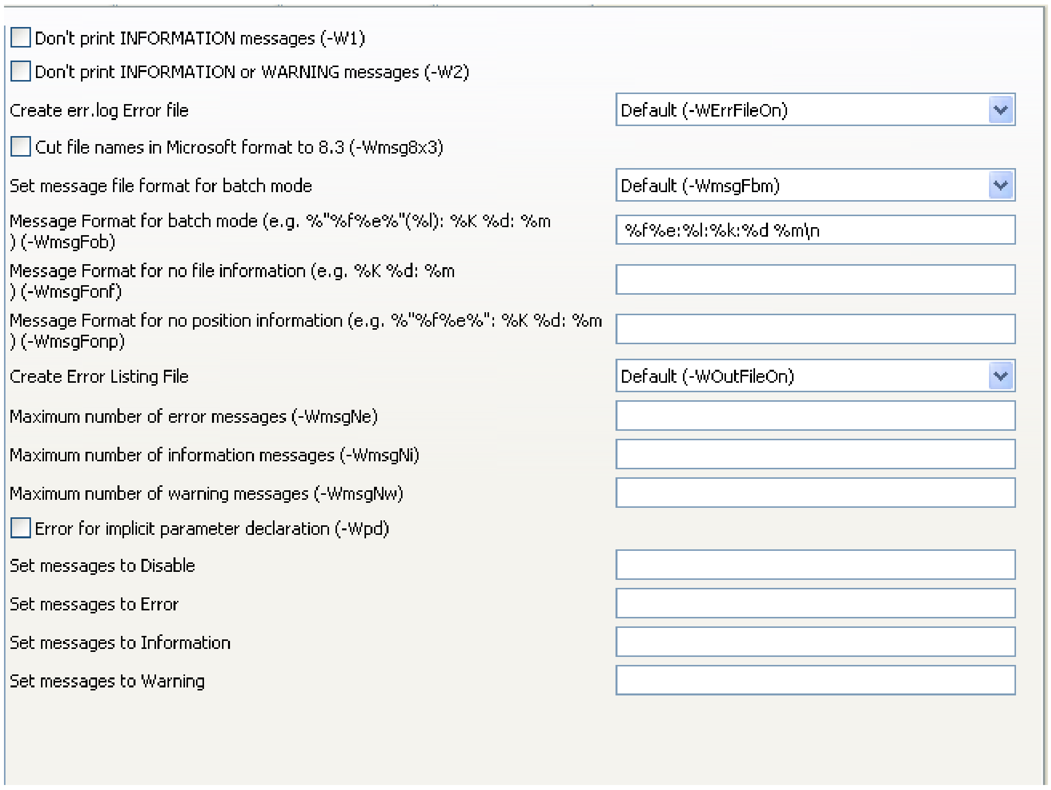

☐ Don't print INFORMATION messages (-W1)
☐ Don't print INFORMATION or WARNING messages (-W2)
Create err.log Error file: Default (-WErrFileOn)
☐ Cut file names in Microsoft format to 8.3 (-Wmsg8x3)
Set message file format for batch mode: Default (-WmsgFbm)
Message Format for batch mode (e.g. %"%f%e%"(%l): %K %d: %m) (-WmsgFob): %f%e:%l:%k:%d %m\n
Message Format for no file information (e.g. %K %d: %m) (-WmsgFonf):
Message Format for no position information (e.g. %"%f%e%": %K %d: %m) (-WmsgFonp):
Create Error Listing File: Default (-WOutFileOn)
Maximum number of error messages (-WmsgNe):
Maximum number of information messages (-WmsgNi):
Maximum number of warning messages (-WmsgNw):
☐ Error for implicit parameter declaration (-Wpd)
Set messages to Disable:
Set messages to Error:
Set messages to Information:
Set messages to Warning:

Figure 3-30. Tool Settings - RS08 Compiler > Messages

The following table lists and describes the messages options for RS08 compiler.

Table 3-12. Tool Settings - RS08 Compiler > Messages Options

Option	Description
Don't print `INFORMATION` messages (`-W1`)	Refer to the -W1: Do Not 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 File Names in Microsoft Format to 8.3 topic.
Set message file format for batch mode	Refer to the -WmsgFb (-WmsgFbi, -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.

Table continues on the next page...

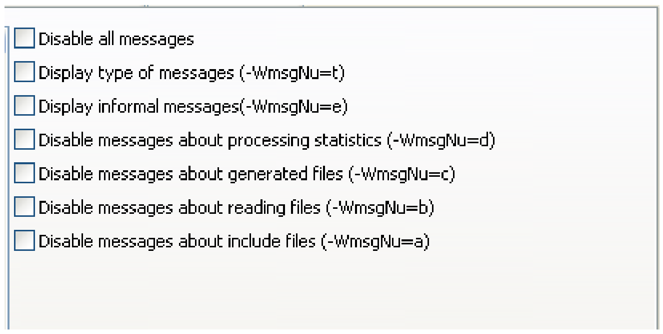
Table 3-12. Tool Settings - RS08 Compiler > Messages Options (continued)

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

3.7.2.11 RS08 Compiler > Messages > Disable user messages

Use this panel to specify the settings to disable the user messages for the RS08 compiler.

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



- [ ] Disable all messages
- [ ] Display type of messages (-WmsgNu=t)
- [ ] Display informal messages(-WmsgNu=e)
- [ ] Disable messages about processing statistics (-WmsgNu=d)
- [ ] Disable messages about generated files (-WmsgNu=c)
- [ ] Disable messages about reading files (-WmsgNu=b)
- [ ] Disable messages about include files (-WmsgNu=a)

Figure 3-31. Tool Settings - RS08 Compiler > Messages > Disable user messages

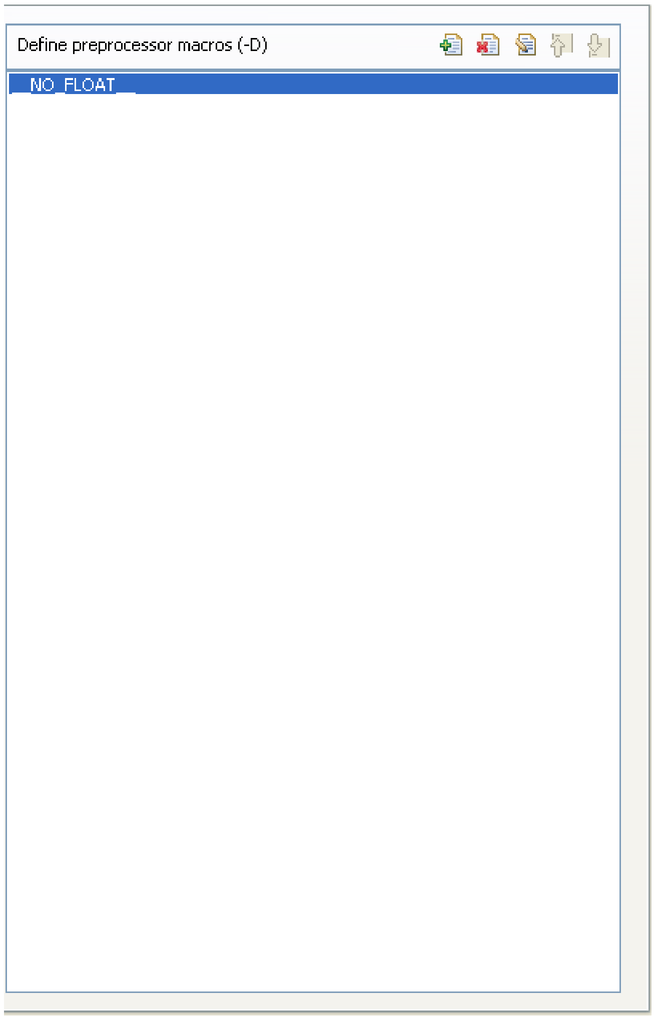
NOTE

For information about the description of the options available in the **Disable user messages** panel for RS08 Compiler, refer to the -WmsgNu: Disable User Messages topic.

3.7.2.12 RS08 Compiler > Preprocessor

Use this panel to specify preprocessor behavior and define macros.

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


Define preprocessor macros (-D)

__NO_FLOAT__

Figure 3-32. Tool Settings - RS08 Compiler > Preprocessor

The following table lists and describes the preprocessor options for RS08 Compiler.

Table 3-13. Tool Settings - RS08 Compiler > Preprocessor Options

Option	Description
Define preprocessor macros (`-D`)	Define, delete, or rearrange preprocessor macros. You can specify multiple macros and change the order in which the IDE uses the macros.

Table 3-13. Tool Settings - RS08 Compiler > Preprocessor Options

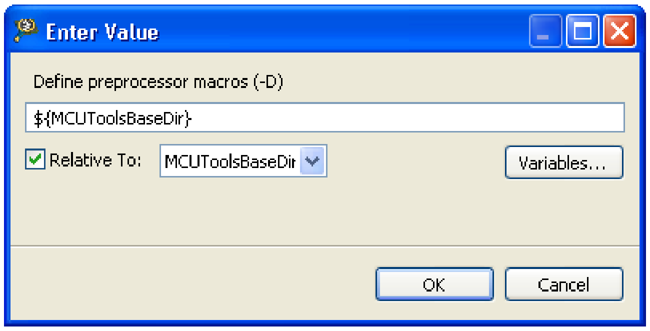
Option	Description
	Define preprocessor macros and optionally assign their values. This setting is equivalent to specifying the `-D name[=value]` command-line option. To assign a value, use the equal sign (=) with no white space. For example, this syntax defines a preprocessor value named `EXTENDED_FEATURE` and assigns `ON` as its value: `EXTENDED_FEATURE=ON` Note that 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. To confirm deletion, click **Yes** in the **Confirm Delete** dialog box.
✎	**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.


Enter Value

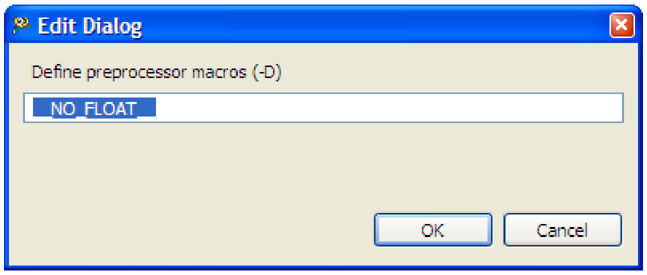
Define preprocessor macros (-D)

${MCUToolsBaseDir}

Relative To: [✓] MCUToolsBaseDir

Figure 3-33. Tool Settings - RS08 Compiler > Preprocessor - Enter Value Dialog Box

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



Define preprocessor macros (-D)

NO_FLOAT

Figure 3-34. Tool Settings - RS08 Compiler > Preprocessor - Edit Dialog Dialog Box

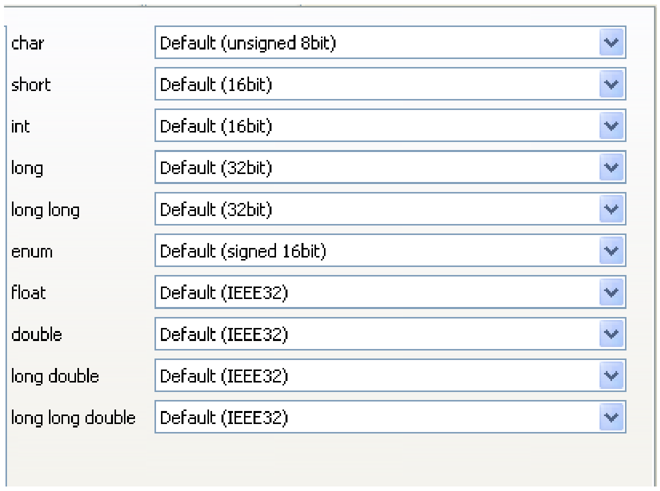
The buttons in the **Enter Value and Edit Dialog** , dialog boxes help work with the preprocessor macros.

- **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 open the **Select build variable** dialog box to select the desired build variable.
- **Relative To -** Check this checkbox to enable the build variables drop-down list. You can select the desired variable for the existing settings.

3.7.2.13 RS08 Compiler > Type Sizes

Use this panel to specify the available data type size options for the RS08 Compiler.

The following figure shows the **Type Sizes** panel.



char	Default (unsigned 8bit)
short	Default (16bit)
int	Default (16bit)
long	Default (32bit)
long long	Default (32bit)
enum	Default (signed 16bit)
float	Default (IEEE32)
double	Default (IEEE32)
long double	Default (IEEE32)
long long double	Default (IEEE32)

Figure 3-35. Tool Settings - RS08 Compiler > Type Sizes

NOTE

For information about the description of the options available in the **Type Sizes** panel for RS08 compiler, refer to the -T: Flexible Type Management topic.

3.7.2.14 RS08 Compiler > General

Use this panel to specify other flags for the RS08 Compiler to use.

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

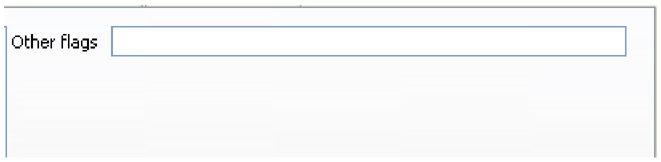


Figure 3-36. Tool Settings - RS08 Compiler > General

The following table lists and describes the General options for RS08 compiler.

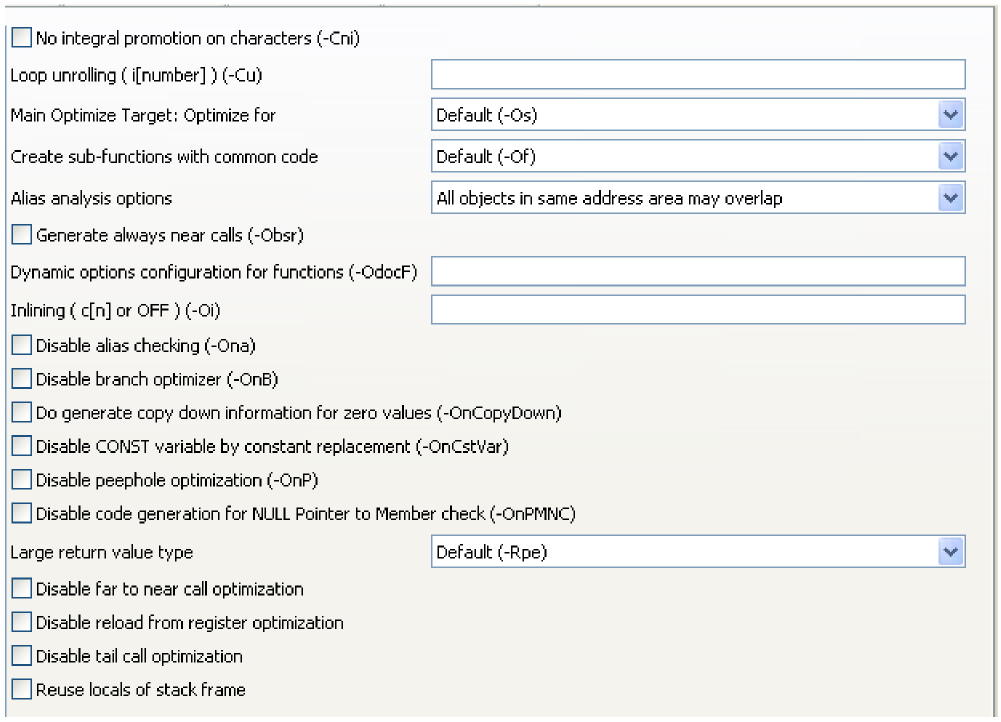
Table 3-15. Tool Settings - RS08 Compiler > General Options

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

3.7.2.15 RS08 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.



- [ ] No integral promotion on characters (-Cni)

Loop unrolling (i[number]) (-Cu)	
Main Optimize Target: Optimize for	Default (-Os)
Create sub-functions with common code	Default (-Of)
Alias analysis options	All objects in same address area may overlap

- [ ] Generate always near calls (-Obsr)

Dynamic options configuration for functions (-OdocF)	
Inlining (c[n] or OFF) (-Oi)	

- [ ] Disable alias checking (-Ona)
- [ ] Disable branch optimizer (-OnB)
- [ ] Do generate copy down information for zero values (-OnCopyDown)
- [ ] Disable CONST variable by constant replacement (-OnCstVar)
- [ ] Disable peephole optimization (-OnP)
- [ ] Disable code generation for NULL Pointer to Member check (-OnPMNC)

Large return value type	Default (-Rpe)

- [ ] Disable far to near call optimization
- [ ] Disable reload from register optimization
- [ ] Disable tail call optimization
- [ ] Reuse locals of stack frame

Figure 3-37. Tool Settings - RS08 Compiler > Optmization

The following table lists and describes the optimization options for RS08 compiler.

Table 3-16. Tool Settings - RS08 Compiler > Optimization Options

Option	Description
No integral promotion on characters (`-Cni`)	Refer to the -Cni: No Integral Promotion 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.
Create sub-functions with common code	Refer to the -O[nf|f]: Create Sub-Functions with Common Code topic.
Alias analysis options	Refer to the -Oa: Alias Analysis Options topic.
Generate always near calls (`-Obsr`)	Refer to the -Obsr: Generate Always Near Calls 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.
Disable branch optimizer (`-OnB`)	Refer to the -OnB: Disable Branch Optimizer 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 peephole optimization (`-OnP`)	Refer to the -Onp: Disable Peephole Optimizer 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[t|e]: Large return Value Type topic.

Table continues on the next page...

Table 3-16. Tool Settings - RS08 Compiler > Optimization Options (continued)

Option	Description
Disable far to near call optimization	Refer to the -Onbsr: Disable far to near call optimization topic.
Disable reload from register optimization	Refer to the -Onr: Disable Reload from Register Optimization topic.
Disable tail call optimizations	Refer to the -Ontc: Disable tail call optimization topic.
Reuse locals of stack frame	Refer to the -Ostk: Reuse Locals of Stack Frame topic.

3.7.2.16 RS08 Compiler > Optimization > Mid level optimizations

Use this panel to configure the Mid level optimization options for the RS08 compiler.

The following figure shows the **Mid level optimizations** settings.

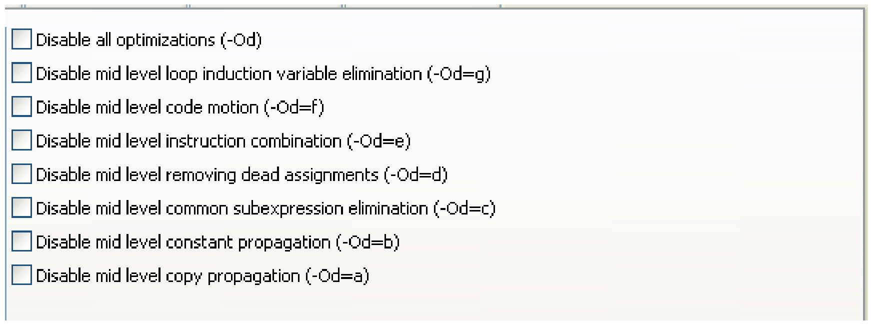

- [ ] Disable all optimizations (-Od)
- [ ] Disable mid level loop induction variable elimination (-Od=g)
- [ ] Disable mid level code motion (-Od=f)
- [ ] Disable mid level instruction combination (-Od=e)
- [ ] Disable mid level removing dead assignments (-Od=d)
- [ ] Disable mid level common subexpression elimination (-Od=c)
- [ ] Disable mid level constant propagation (-Od=b)
- [ ] Disable mid level copy propagation (-Od=a)

Figure 3-38. Tool Settings - RS08 Compiler > Optimization > Mid level optimizations

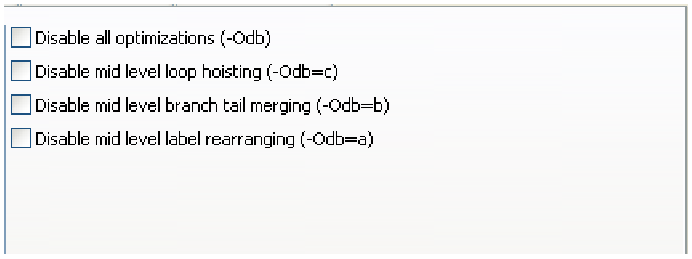
NOTE

For information about the description of the options available in the **Mid level optimizations** panel for RS08 compiler, refer to the -Od: Disable Mid-level Optimizations topic.

3.7.2.17 RS08 Compiler > Optimization > Mid level branch optimizations

Use this panel to configure the Mid level branch optimization options for the RS08 compiler.

The following figure shows the **Mid level branch optimizations** settings.



- [ ] Disable all optimizations (-Odb)
- [ ] Disable mid level loop hoisting (-Odb=c)
- [ ] Disable mid level branch tail merging (-Odb=b)
- [ ] Disable mid level label rearranging (-Odb=a)

Figure 3-39. Tool Settings - RS08 Compiler > Optimization > Mid level branch optimizations

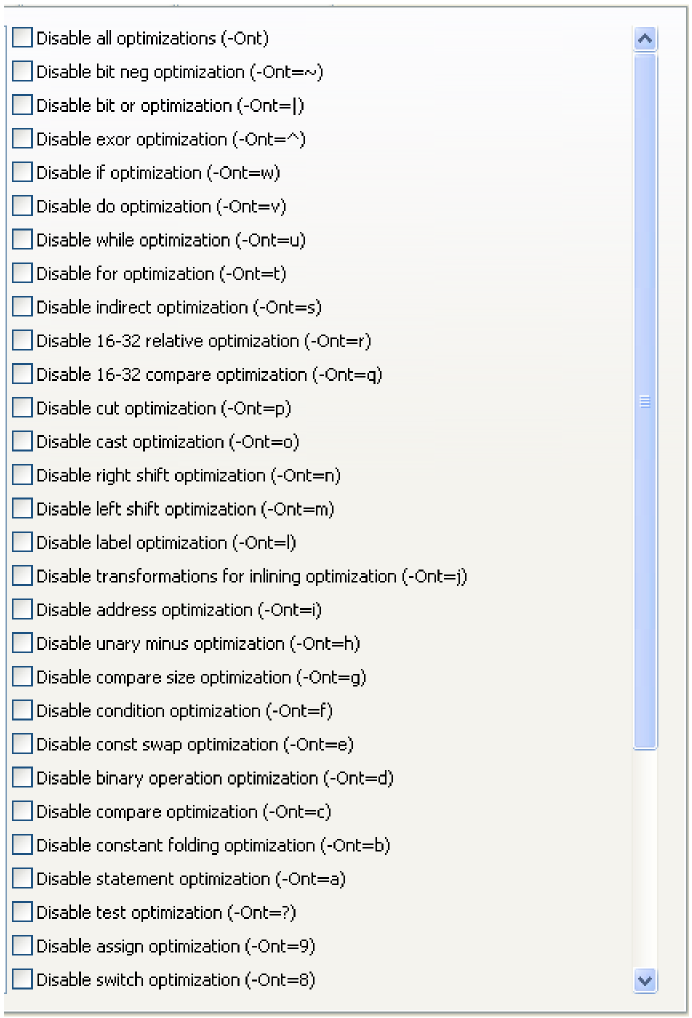
NOTE

For information about the description of the options available for the **Mid levelbranch optimizations** compiler settings, refer to the -Odb: Disable Mid-level Branch Optimizations topic.

3.7.2.18 RS08 Compiler > Optimization > Tree optimizer

Use this panel to configure the tree optimizer options for the RS08 compiler.

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



- [ ] Disable all optimizations (-Ont)
- [ ] Disable bit neg optimization (-Ont=~)
- [ ] Disable bit or optimization (-Ont=|)
- [ ] Disable exor optimization (-Ont=^)
- [ ] Disable if optimization (-Ont=w)
- [ ] Disable do optimization (-Ont=v)
- [ ] Disable while optimization (-Ont=u)
- [ ] Disable for optimization (-Ont=t)
- [ ] Disable indirect optimization (-Ont=s)
- [ ] Disable 16-32 relative optimization (-Ont=r)
- [ ] Disable 16-32 compare optimization (-Ont=q)
- [ ] Disable cut optimization (-Ont=p)
- [ ] Disable cast optimization (-Ont=o)
- [ ] Disable right shift optimization (-Ont=n)
- [ ] Disable left shift optimization (-Ont=m)
- [ ] Disable label optimization (-Ont=l)
- [ ] Disable transformations for inlining optimization (-Ont=j)
- [ ] Disable address optimization (-Ont=i)
- [ ] Disable unary minus optimization (-Ont=h)
- [ ] Disable compare size optimization (-Ont=g)
- [ ] Disable condition optimization (-Ont=f)
- [ ] Disable const swap optimization (-Ont=e)
- [ ] Disable binary operation optimization (-Ont=d)
- [ ] Disable compare optimization (-Ont=c)
- [ ] Disable constant folding optimization (-Ont=b)
- [ ] Disable statement optimization (-Ont=a)
- [ ] Disable test optimization (-Ont=?)
- [ ] Disable assign optimization (-Ont=9)
- [ ] Disable switch optimization (-Ont=8)

Figure 3-40. Tool Settings - RS08 Compiler > Optimization > Tree optimizer

NOTE

For information about the description of the options available in the **Tree optimizer** panel for RS08 compiler, refer to the -Ont: Disable Tree Optimizer topic.

3.7.2.19 RS08 Compiler > Optimization > Optimize Library Function

Use this panel to configure the optimize library function options for the RS08 compiler.

The following figure shows the **Optimize Library Function** settings.

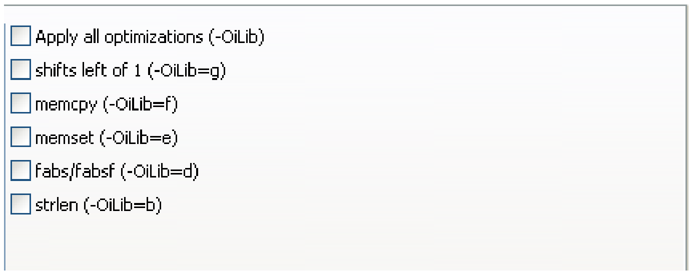

- [ ] Apply all optimizations (-OiLib)
- [ ] shifts left of 1 (-OiLib=g)
- [ ] memcpy (-OiLib=f)
- [ ] memset (-OiLib=e)
- [ ] fabs/fabsf (-OiLib=d)
- [ ] strlen (-OiLib=b)

Figure 3-41. Tool Settings - RS08 Compiler > Optimization > Optimize Library Function

NOTE

For information about the description of the options available for the **OptimizeLibrary Function** panel for RS08 compiler, refer to the -OiLib: Optimize Library Functions topic.

3.7.3 CodeWarrior Tips and Tricks

If the Simulator or Debugger cannot be launched, check the settings in the *project's launch configuration*. For more information on 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 **RunConfigurations** 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:

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

   The **Properties for <** project **>** window appears.
2. Expand **C/C++ General** and select **File Types**.
3. Select the **Use project settings** option.
4. Click the **New** button, enter the required file type, and click **OK**.
5. Click **OK.**

The changes are saved and the properties dialog box closes.

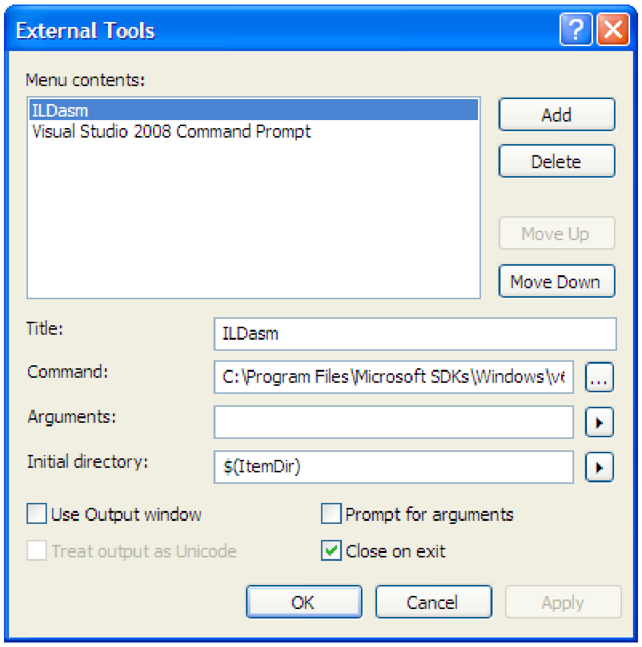
3.8 Integration into Microsoft Visual C++ 2008 Express Edition (Version 9.0 or later)

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

3.8.1 Integration as External Tools

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

   The **External Tools** dialog box appears.



Title: ILDasm
Command: C:\Program Files\Microsoft SDKs\Windows\v€
Arguments:
Initial directory: $(ItemDir)
[✓ Close on exit]

Figure 3-42. External Tools Dialog Box

3. Add a new tool by clicking *Add*.
4. In the **Title** text box, type in the name of the tool to display in the menu, for example, `RS08 Compiler`.
5. In the *Command* text box, either type the path of the `piper` tool or browse to `Piper.exe` using the " **...** " 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 RS08 build tools can be operated from within the Visual Studio UI.

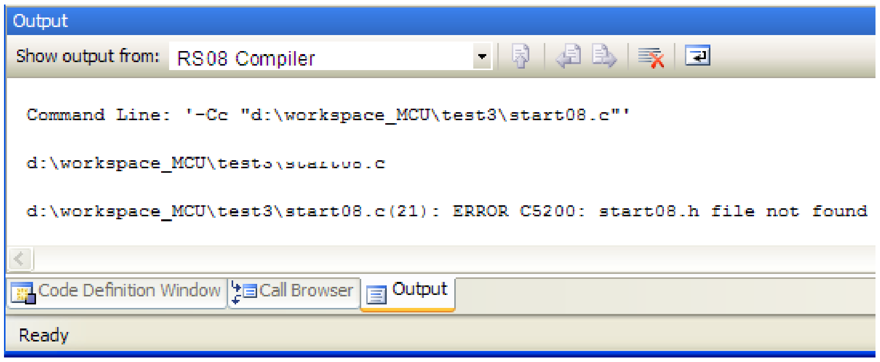
6. In the *Arguments* text box, type the name of the tool to be launched. You can also specify any command line options, for example, `-Cc`, along with the `$(ItemPath)Visual` variable.

   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\crs08.exe -Cc $(ItemPath)`.
7. In the **Initial Directory** text box, use the pop-up menu to choose `$(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)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 on the error message in this window and the offending source code statement is displayed in the editor.



```
Command Line: '-Cc "d:\workspace_MCU\test3\start08.c"'

d:\workspace_MCU\test3\start08.c

d:\workspace_MCU\test3\start08.c(21): ERROR C5200: start08.h file not found
```

Figure 3-43. Visual Output Window

3.8.2 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 on the **Tools** menu.
3. Select **Tools > Customize**.

The **Customize** window appears.

4. Click 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. Click 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** field 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 **RS08 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 RS08 Linker and the RS08 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** in 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 appear in the toolbar.
16. Click **Close** to close the **Customize** window.

This enables you to start the CodeWarrior tools from the toolbar.

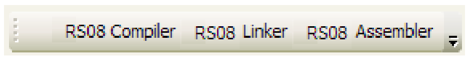


Figure 3-44. CodeWarrior Tools in Custom Visual Studio Toolbar

3.9 Object-File Formats

The compiler supports only the ELF/DWARF object-file format. The object-file format specifies the format of the object files (`*.o` extension), the library files (`*.lib` extension), and the absolute files (`*.abs` extension).

3.9.1 ELF/DWARF Object-File Format

The ELF/DWARF object-file format originally comes from the UNIX world. This format is very flexible and is able to support 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 tool in the tool chain. In addition, other third parties (for example, emulator vendors) only have to support this object file to support a wide range of tool vendors.

3.9.2 Mixing Object-File Formats

Mixing HIWARE and ELF object files is not possible. HIWARE object file formats are not supported on the RS08. Mixing ELF object files with DWARF 1.1 and DWARF 2.0 debug information is possible. However, the final generated application does not contain any debug data.

NOTE

Be careful when mixing object files made with the HCS08 compiler, which can generate an alternate object file format called HIWARE. Both HIWARE and the ELF/DWARF object files use the same filename extensions. The HCS08 compiler must be configured to generate the ELF/DWARF object format. For information about how to do this, refer to the **Microcontrollers V10.x HC08 Build Tools Reference Manual**.

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 RS08 Compiler:

- Navigate to `CWInstallDir\MCU\prog` using Windows Explorer and double-click `crs08.exe`

- Create a shortcut for `crs08.exe` on the desktop and double-click the shortcut
- Create a shortcut for `crs08.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.

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 RS08 compiler and process two source files, `a.c` and `d.c`, which appear as arguments:

```
C:\Freescale\CW MCU V10.x\MCU\prog>crs08.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>crs08.exe - F2 a.c d.c > myoutput.o
```

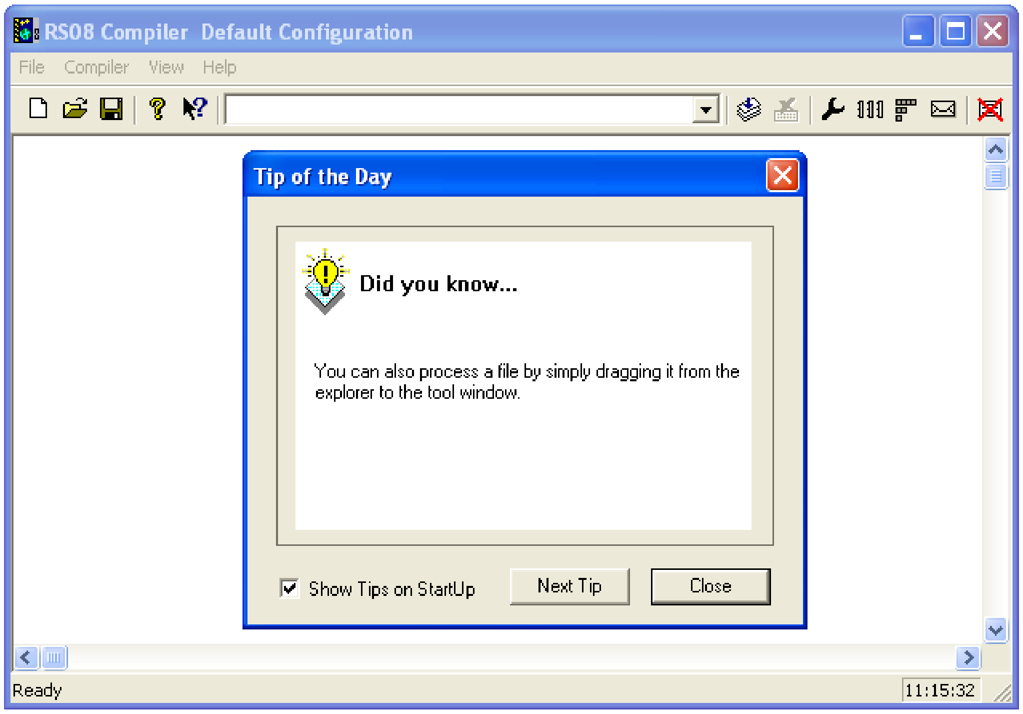
The command line process returns after the compiling process starts. It does not wait until the previous process has terminated. To start a tool and wait for termination , for example, to synchronize successive operations, use the `start` command in the Windows® 2000, Windows XP, or Windows Vista™ operating 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 crs08.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.



Did you know...

You can also process a file by simply dragging it from the explorer to the tool window.

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.

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).

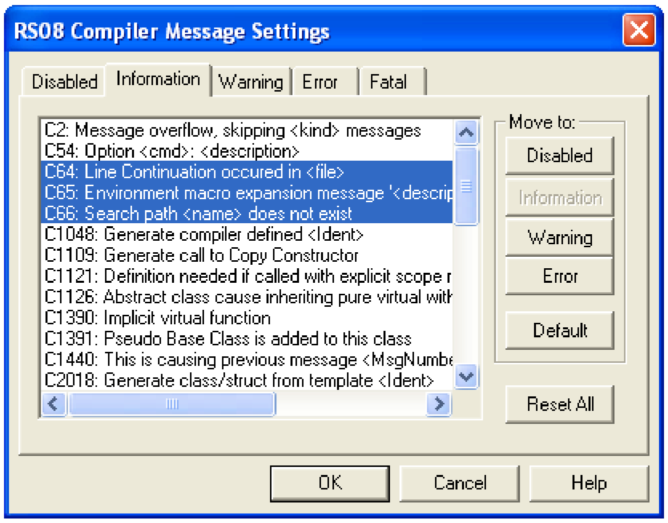
The message number is available for any message output. There are three ways to open the corresponding entry in the help file.

- 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 **RS08 Compiler Message Settings Dialog Box** .



```
C2: Message overflow, skipping <kind> messages
C54: Option <cmd>: <description>
C64: Line Continuation occured in <file>
C65: Environment macro expansion message '<descrip
C66: Search path <name> does not exist
C1048: Generate compiler defined <Ident>
C1109: Generate call to Copy Constructor
C1121: Definition needed if called with explicit scope r
C1126: Abstract class cause inheriting pure virtual with
C1390: Implicit virtual function
C1391: Pseudo Base Class is added to this class
C1440: This is causing previous message <MsgNumbe
C2018: Generate class/struct from template <Ident>
```

Figure 4-2. RS08 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, as shown in the figure below, 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`.

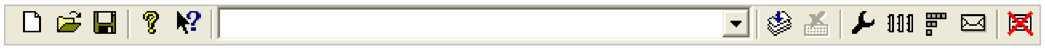
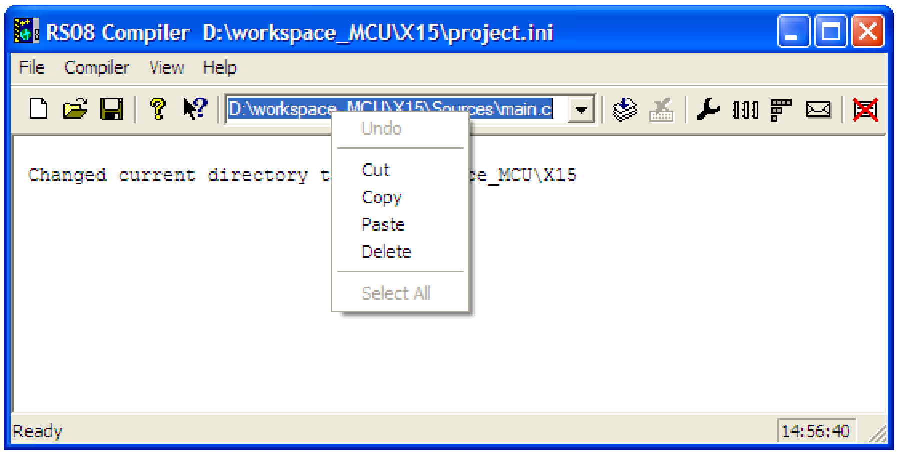


Figure 4-3. Toolbar

- 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 as shown in the figure below.
- 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).



RS08 Compiler D:\workspace_MCU\X15\project.ini

File Compiler View Help

D:\workspace_MCU\X15\Sources\main.c

```
Changed current directory t          e_MCU\X15
```

Undo

Cut
Copy
Paste
Delete

Select All

Ready 14:56:40

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.

Figure 4-5. Status Bar

4.2.5 Compiler Menu Bar

The following table lists and describes the menus available in the menu bar as shown in the figure below:

Table 4-1. Menus in the Menu Bar

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

Figure 4-6. Menu Bar

4.2.6 File Menu

The **File** menu is used to save or load configuration files (Figure 4-7). A 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

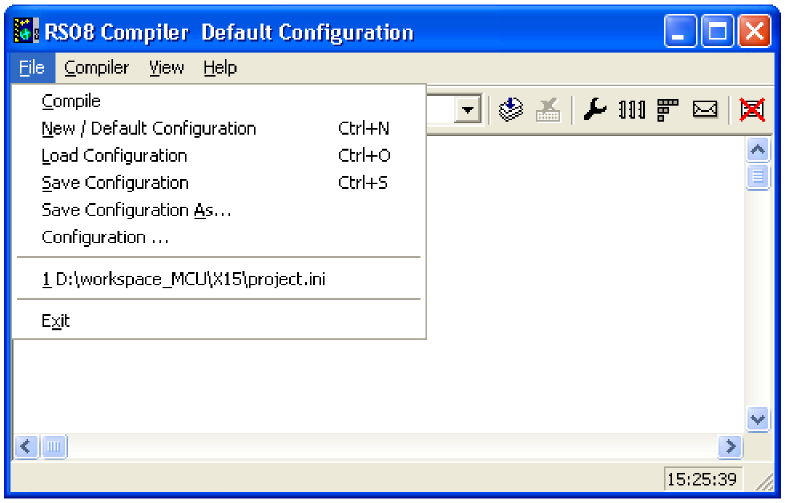

RS08 Compiler Default Configuration
File Compiler View Help
Compile
New / Default Configuration Ctrl+N
Load Configuration Ctrl+O
Save Configuration Ctrl+S
Save Configuration As...
Configuration ...
1 D:\workspace_MCU\X15\project.ini
Exit
15:25:39

Figure 4-7. File Menu

Configuration files are text files which use the standard extension `*.ini`. A developer can define as many configuration files as required for a project. The developer can also switch between the different configuration files using the **File > Load Configuration** and **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 Editor Settings Dialog Box and Save Configuration Dialog Box topics.
1... project.ini 2...	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 following table lists the **Compiler** menu options.

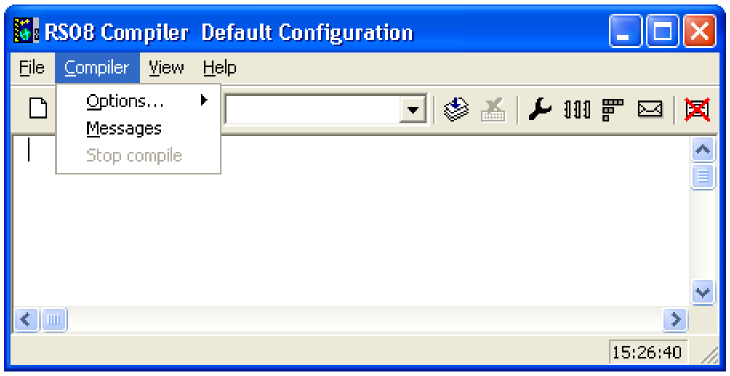


Figure 4-8. Compiler Menu

Table 4-3. Compiler Menu Options

Menu Option	Description
Options	Allows you to customize the application. You can graphically set or reset options. The next three entries are available when **Options** is selected:
Standard Types	Allows you to specify the size you want to associate with each ANSI C standard type. (Refer to the Standard Types Settings Dialog Box topic.)
Options	Allows you to define the options which must be activated when processing an input file. (See Option Settings Dialog Box.).
Smart Slider	Allows you to define the optimization level you want to reach when processing the input file. (See Smart Control Dialog Box.)
Messages	Opens a dialog box, where the different error, warning, or information messages are mapped to another message class. (See Message Settings Dialog Box.)
Stop Compile	Immediately stops the current processing session.

4.2.8 View Menu

The **View** menu enables you to customize the application window. You can define things such as displaying or hiding the status or toolbar. You can also define the font used in the window, or clear the window. The following image shows the **View** Menu options.

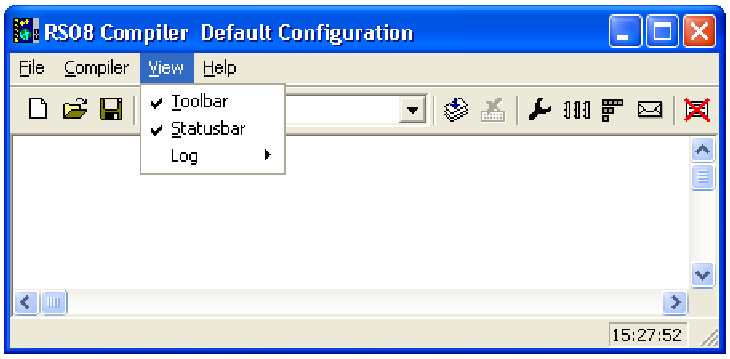


Figure 4-9. View Menu

The following table lists the **View** Menu options.

Table 4-4. View Menu Options

Menu Option	Description
Toolbar	Switches display from the toolbar in the application window.
Status Bar	Switches display from the status bar in the application window.
Log	Allows you to customize the output in the application window content area. The following entries are available when **Log** is selected:
Change Font	Opens a standard font selection box. The options selected in the font dialog box are applied to the application window content area.
Clear Log	Allows you to clear the application window content area.

4.2.9 Help Menu

The **Help** Menu enables you to either display or hide the Tip of the Day dialog box at application startup. In addition, it provides a standard Windows Help entry and an entry to an About box. The following image shows the **Help** Menu.

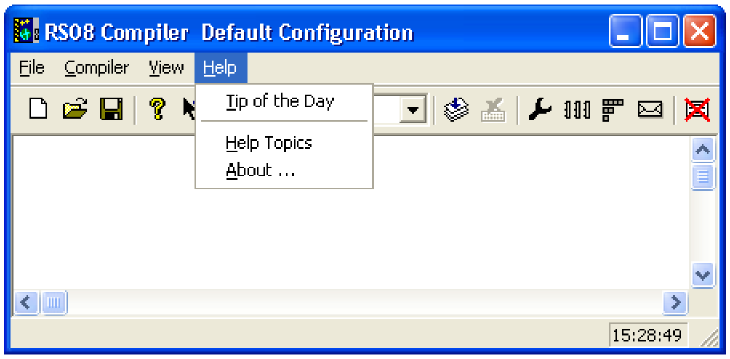


Figure 4-10. Help Menu

The following table lists the **Help** Menu options.

Table 4-5. Help Menu Options

Menu Option	Description
Tip of the Day	Switches on or off the display of a Tip of the Day during startup.
Help Topics	Standard Help topics.
About	Displays an About box with some version and license information.

4.3 Editor Settings Dialog Box

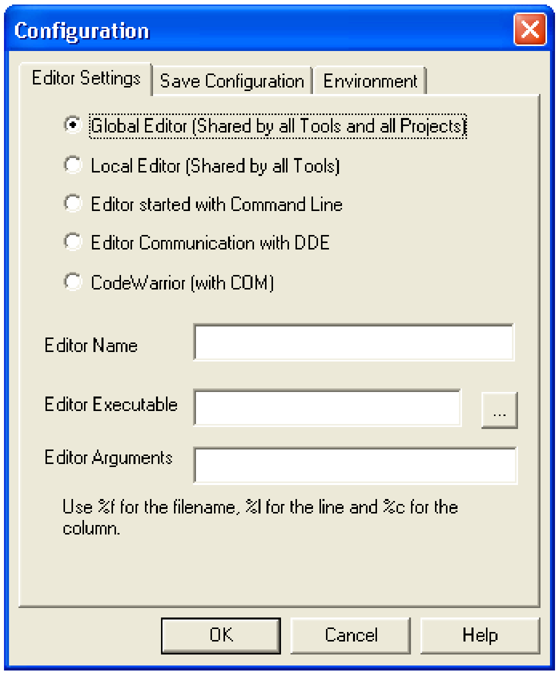
The **Editor Setting***s* dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

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)

4.3.1 Global Editor (shared by all tools and projects)

The Global Editor is shared among all tools and projects on one work station. It is stored in the global initialization file `mcutools.ini` in the `[Editor]` section. Some Modifiers are specified in the editor command line.



[✓ Global Editor (Shared by all Tools and all Projects)]
Local Editor (Shared by all Tools)
Editor started with Command Line
Editor Communication with DDE
CodeWarrior (with COM)

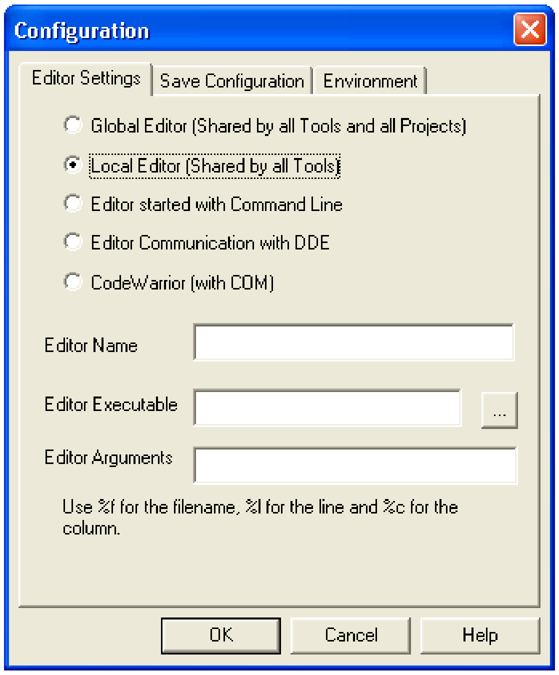
Editor Name
Editor Executable
Editor Arguments

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

Figure 4-11. Global Editor Configuration Dialog Box

4.3.2 Local Editor (shared by all tools)

The Local Editor is shared among all tools using the same project file. When an entry of the Global or Local configuration is stored, the behavior of the other tools using the same entry also changes when these tools are restarted.


Configuration

Editor Settings | Save Configuration | Environment

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

Editor Name

Editor Executable

Editor Arguments

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

Figure 4-12. Local Editor configuration Dialog Box

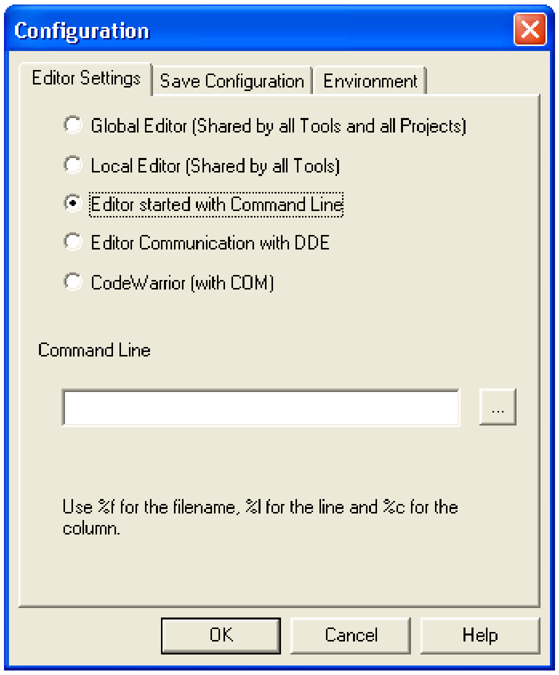
4.3.3 Editor Started with Command Line

When this editor type, as shown in the figure below, is selected, a separate editor is associated with the application for error feedback. The configured editor is not used for error feedback.

Enter the command that starts the editor.

The format of the editor command depends on the syntax. Some Modifiers are specified in the editor command line to refer to a line number in the file. (For more information, refer to the Modifiers section of this manual.)

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



Global Editor (Shared by all Tools and all Projects)
Local Editor (Shared by all Tools)
[✓ Editor started with Command Line]
Editor Communication with DDE
CodeWarrior (with COM)

Command Line

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

Figure 4-13. Editor Started with Command Line

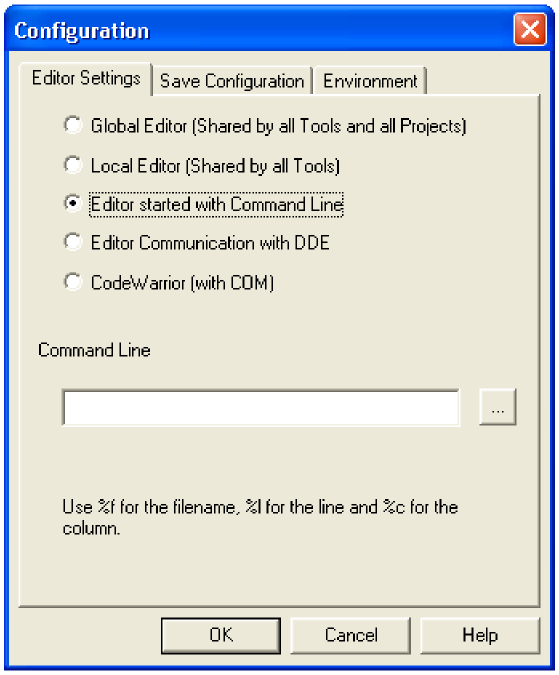
4.3.3.1 Examples:

For CodeWright V6.0 version, use (with an adapted path to the `cw32.exe` file): `C:\CodeWright\cw32.exe %f -g%l`

For the WinEdit 32-bit version, use (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 (Microsoft Developer Studio [Figure 4-14] or UltraEdit-32 [Figure 4-15]). The **Topic Name** and **Client Command** entries can have modifiers for the filename, line number, and column number as explained in Modifiers.



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

Command Line

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

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)]
```

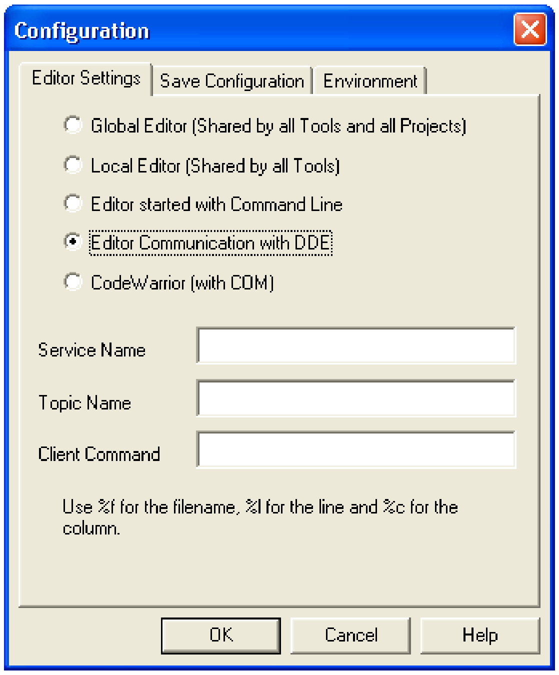
UltraEdit-32 is a powerful shareware editor. It is available from `www.idmcomp.com`www.idmcomp.comwww.idmcomp.com or www.ultraedit.com, email idm@idmcomp.com. For UltraEdit, use the following settings.

Listing: UltraEdit-32 editor settings.

```
Service Name : UEDIT32
Topic Name : system
Client Command : [open("%f/%l/%c")]
```

NOTE

The DDE application (e.g., Microsoft Developer Studio or UltraEdit) must be started or the DDE communication fails.



( ) Global Editor (Shared by all Tools and all Projects)
( ) Local Editor (Shared by all Tools)
( ) Editor started with Command Line
[✓] Editor Communication with DDE
( ) CodeWarrior (with COM)

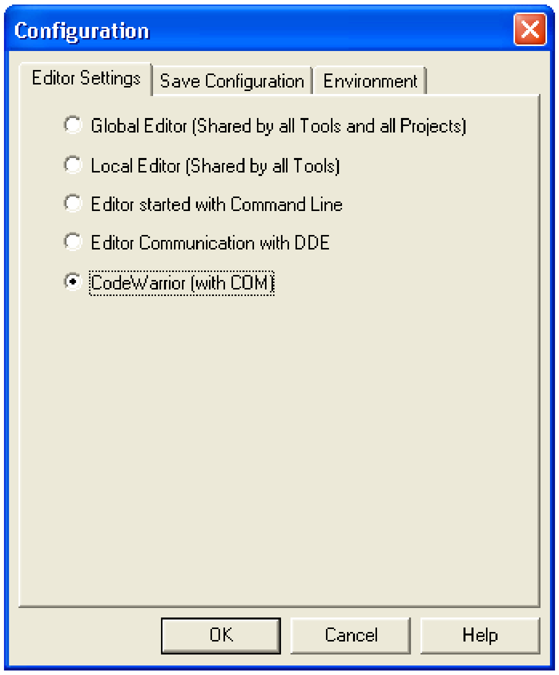
Service Name
Topic Name
Client Command

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

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

4.3.5 CodeWarrior (with COM)

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



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

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.

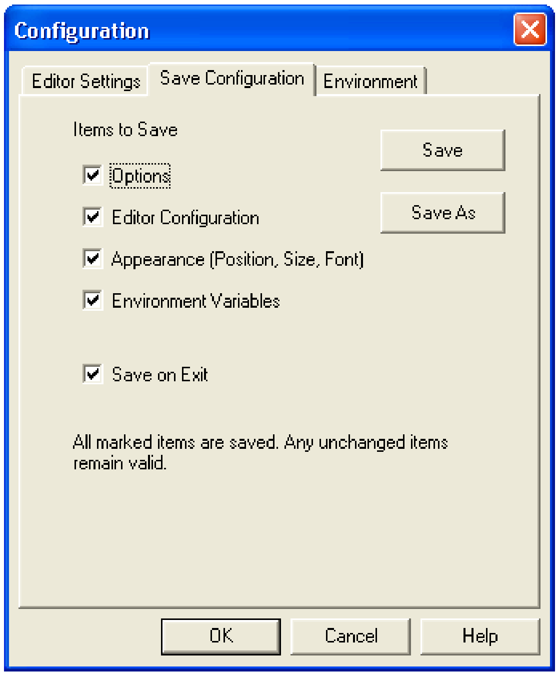
- The `%f` modifier refers to the name of the file (including path) where the message has been detected.
- The `%l` modifier refers to the line number where the message has been detected.
- The `%c` modifier refers to the column number where the message has been detected.

NOTE

The `%l` modifier can only be used with an editor which is started with a line number as a parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When working with such an editor, start it with the filename as a parameter and then select the menu entry **Go to** to jump on the line where the message has been detected. In that case the editor command looks like, `C:\WINAPPS\WINEDIT\Winedit.EXE %f` Check the editor manual to define which command line to use to start the editor.

4.4 Save Configuration Dialog Box

Use the **Save Configuration** dialog box to configure which parts of your configuration you want to save to the project file.



Items to Save

- [✓] Options
- [✓] Editor Configuration
- [✓] Appearance (Position, Size, Font)
- [✓] Environment Variables
- [✓] Save on Exit

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

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 disabling selective options 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 will no longer modify the options.

- Save on Exit

  The application writes the configuration on exit. No question dialog box appears to confirm this operation. If this option is not set, the application will not write the configuration at exit, even if options or another part of the configuration have changed. No confirmation appears in either case when closing the application.

  NOTE

  Most settings are stored in the configuration file only. The only exceptions are, the recently used configuration list, and all settings in this dialog box.

  NOTE

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

4.5 Environment Configuration Dialog Box

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

The following environment variables are available:

Listing: Environment variables

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

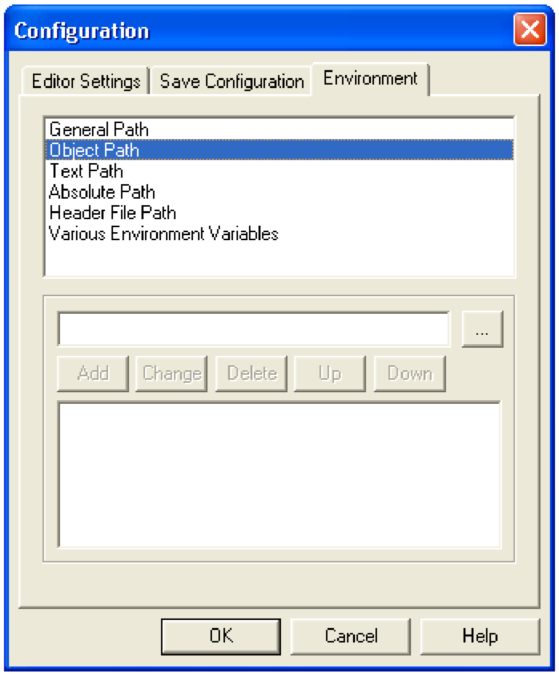


Figure 4-18. Environment Configuration Dialog Box

The buttons available in this dialog box are listed in the following table:

Table 4-6. Environment Configuration Dialog Box 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 that the variables are written to the project file only if you press the **Save** button, select **File -> Save Configuration** or press CTRL-S. In addition, it can be specified in the **Save Configuration** dialog box if the environment is written to the project file or not.

4.6 Standard Types Settings Dialog Box

The **Standard Types Settings** dialog box (Standard Types Settings Dialog Box) enables you to define the size you want to associate to 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 has to be at least one type for each size. For example, it is incorrect to specify all types to a size of 32 bits. There is no type for 8 bits and 16 bits available for the Compiler.

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

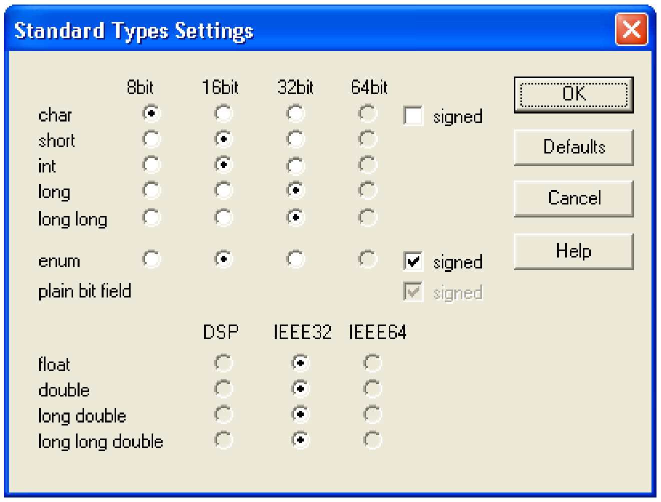
Listing: Size relationships for the ANSI-C standard types

```
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`.

The *signed* check box enables you to specify whether the `char` type must be considered as signed or unsigned for your application.

The *Default* 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.



Standard Types Settings

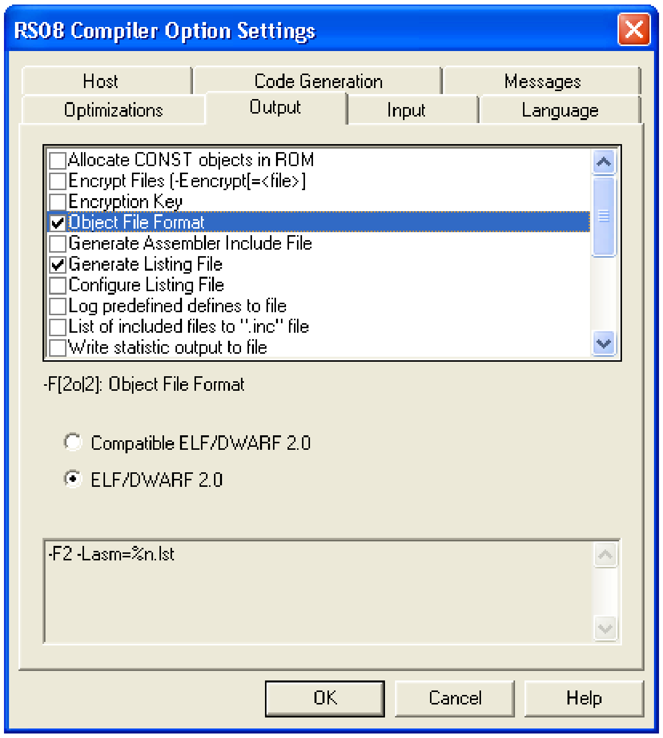
	8bit	16bit	32bit	64bit	
char	●	○	○	○	☐ signed
short	○	●	○	○	
int	○	●	○	○	
long	○	○	●	○	
long long	○	○	●	○	
enum	○	●	○	○	☑ signed
plain bit field					☑ signed

	DSP	IEEE32	IEEE64
float	○	●	○
double	○	●	○
long double	○	●	○
long long double	○	●	○

Figure 4-19. Standard Types Settings Dialog Box

4.7 Option Settings Dialog Box

The **Option Settings** dialog box enables you to set or reset application options. The possible command line option is also displayed in the lower display area The available options are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheet (not all groups may be available). The following table lists the Option Settings dialog box selections.



- [ ] Allocate CONST objects in ROM
- [ ] Encrypt Files (-Eencrypt[=<file>]
- [ ] Encryption Key
- [x] Object File Format
- [ ] Generate Assembler Include File
- [x] Generate Listing File
- [ ] Configure Listing File
- [ ] Log predefined defines to file
- [ ] List of included files to ".inc" file
- [ ] Write statistic output to file

-F[2ol2]: Object File Format

- ○ Compatible ELF/DWARF 2.0
- ● ELF/DWARF 2.0

```
-F2 -Lasm=%n.lst
```

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 files generation (which kind of file to generate).
Input	Lists options related to the input file.
Language	Lists options related to the programming language (ANSI-C)
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 (such as memory models or float format).
Messages	Lists options controlling the generation of error messages.
Various	Lists options not related to 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 options requiring additional parameters are selected, you can open an edit box or an additional sub window where the additional parameter is set. For example, for the option *Write statistic output to file* available in the Output sheet.

4.8 Smart Control Dialog Box

The **Compiler Smart Control** Dialog Box enables you to define the optimization level you want to reach during compilation of the input file. Five sliders are available to define the optimization level. Refer to the table Table 4-8.

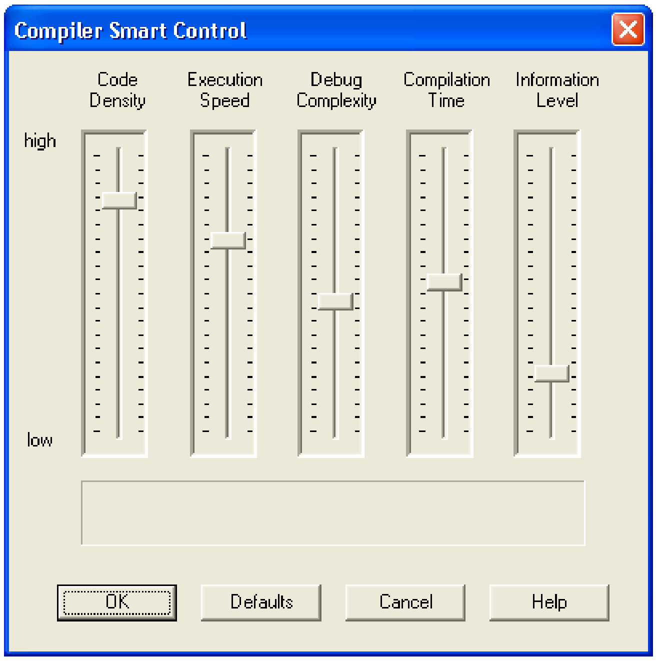


Figure 4-21. Compiler Smart Control Dialog Box

Table 4-8. Compiler Smart Control Dialog Box Controls

Slider	Description
Code Density	Displays the code density level expected. A high value indicates highest code efficiency (smallest code size).
Execution Speed	Displays the execution speed level expected. A high value indicates fastest execution of the code generated.

Table continues on the next page...

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

Slider	Description
Debug Complexity	Displays the debug complexity level expected. A high value indicates complex debugging. For example, assembly code corresponds directly to the high-level language code.
Compilation Time	Displays the compilation time level expected. A higher value indicates longer compilation time to produce the object file, e.g., due to high optimization.
Information Level	Displays the level of information messages which are displayed during a Compiler session. A high value indicates a verbose behavior of the Compiler. For example, it will inform with warnings and information messages.

There is a direct link between the first four sliders in this window. When you move one slider, the positions of the other three are updated according to the modification.

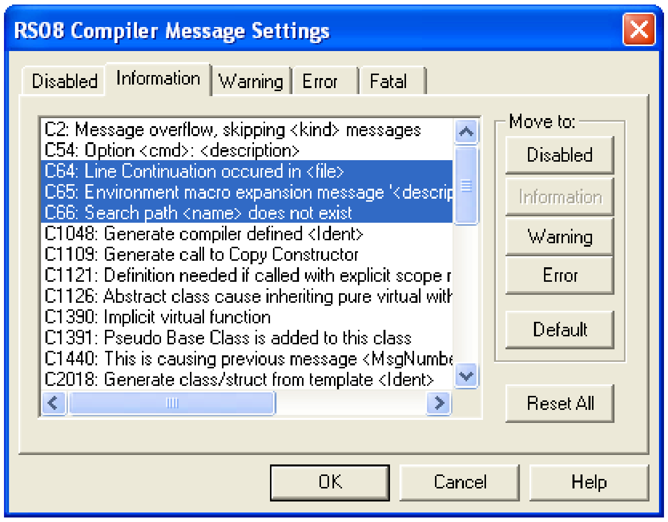
The command line is automatically updated with the options set in accordance with the settings of the different sliders.

4.9 Message Settings Dialog Box

The **Message Settings** dialog box enables you to map messages to a different message class.

Some buttons in the dialog box may be disabled. (For example, if an option cannot be moved to an Information message, the ` **Move to: Information** ' button is disabled.)

The table below lists and describes the buttons available in the **Message Settings** dialog box.



C2: Message overflow, skipping <kind> messages
C54: Option <cmd>: <description>
C64: Line Continuation occured in <file>
C65: Environment macro expansion message '<descrip
C66: Search path <name> does not exist
C1048: Generate compiler defined <Ident>
C1109: Generate call to Copy Constructor
C1121: Definition needed if called with explicit scope r
C1126: Abstract class cause inheriting pure virtual with
C1390: Implicit virtual function
C1391: Pseudo Base Class is added to this class
C1440: This is causing previous message <MsgNumbe
C2018: Generate class/struct from template <Ident>

Figure 4-22. Message Settings Dialog Box

Table 4-9. Message Settings Dialog Box Buttons

Button	Description
Move to: Disabled	The selected messages are disabled. The message will not occur any longer.
Move to: Information	The selected messages are changed to information messages.
Move to: Warning	The selected messages are changed to warning messages.
Move to: Error	The selected messages are changed to error messages.
Move to: Default	The selected messages are changed to their default message type.
Reset All	Resets all messages to their default message kind.
OK	Exits this dialog box and accepts the changes made.
Cancel	Exits this dialog box without accepting the changes made.
Help	Displays online help about this dialog box.

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 definitions for the message groups.

Table 4-10. Message Group Definitions

Message group	Description
Disabled	Lists all disabled messages. That means messages displayed in the list box will not be displayed by the application.
Information	Lists all information messages. Information messages inform about action taken by the application.
Warning	Lists all warning messages. When a warning message is generated, processing of the input file continues.
Error	Lists all error messages. When an error message is generated, processing of the input file 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, processing of the input file stops immediately. Fatal messages cannot be changed and are only listed to call context help.

Each message has its own prefix (e.g., `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.

4.9.1 Changing the Class associated with a Message

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

1. Click the **Warning** panel to display the list of all warning messages in the list box.
2. Click on the message you want to change in the list box to select the message.
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 one message is marked which cannot be moved to a specific group, the corresponding *Move to* button is disabled (grayed).

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

4.9.2 Retrieving Information about an 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 the *F1* key. An information box is opened. The information box 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, a help for the first is shown. When no message is selected, pressing the F1 key or the help button shows the help for this dialog box.

4.10 About Dialog Box

The **About** dialog box is opened by selecting **Help>About.** The About box contains information regarding your application. The current directory and the versions of subparts of the application are also shown. The main version is displayed separately on top of the dialog box.

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 sub-versions of the application parts cannot be requested. They are only displayed if the application is inactive.

4.11 Specifying Input File

There are different ways to specify the input file. During the compilation, the options are set according to the configuration established in the different dialog boxes.

Before starting to compile a file make sure you have associated a working directory with your editor.

4.11.1 Compiling Using Command Line in Toolbar

The command line can be used to compile a new file and to open a file that has already been compiled.

4.11.1.1 Compiling New File

A new filename and additional Compiler options are entered in the command line. The specified file is compiled as soon as the *Compile* button in the toolbar is selected or the Enter key is pressed.

4.11.1.2 Compiling Already Compiled File

The previously executed command is displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file is compiled as soon as the *Compile* button in the toolbar is clicked.

4.11.1.3 Using File > Compile Menu Entry

When the menu entry *File* > *Compile* is selected, a standard open file box is displayed. Use this to locate the file you want to compile. The selected file is compiled as soon as the standard open file box is closed using the *Open* button.

4.11.1.4 Using Drag and Drop

A filename is dragged from an external application (for example the File Manager/ Explorer) and dropped into the Compiler window. The dropped file is compiled as soon as the mouse button is released in the Compiler window. If a file being dragged has the `*.ini` extension, it is considered to be a configuration and it is immediately loaded and not compiled. To compile a C file with the `*.ini` extension, use one of the other methods.

4.11.2 Message/Error Feedback

There are several ways to check where different errors or warnings have been detected after compilation. 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 is a typical example of an 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
```

See also the -WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode and -WmsgFb (-WmsgFbi, -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 has been compiled, the Compiler window content area displays the list of all the errors or warnings that were detected.

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

4.11.4 Use User-Defined Editor

You must first configure the editor you want to use for message/error feedback in the **Configuration** dialog box before you begin the compile process. Once a file has been compiled, double-click on an error message. The selected editor is automatically activated and points to the line containing the error.

Chapter 5
Environment

This Chapter describes all the environment variables. Some environment variables are also used by other tools (For example, Linker or Assembler). Consult the respective manual for more information.

Parameters are set in an environment using environment variables. There are three ways to specify your environment:

- The current project file with the [Environment Variables] section. This file may be specified on Tool startup using the -Prod: Specify Project File at Startup option.
- An optional `default.env' file in the current directory. This file is supported for backwards compatibility. The filename is specified using the ENVIRONMENT: Environment File Specification variable. Using the default.env file is not recommended.
- Setting environment variables on system level (DOS level). This is not recommended.

The syntax for setting an environment variable is (Listing: Setting the GENPATH 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
```

Parameters may be defined in several ways:

- Using system environment variables supported by your operating system.

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

The default directory mentioned above is set using the DEFAULTDIR: Default Current Directory system environment variable.

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, a default value is assumed.

NOTE

The environment may also be changed using the -Env: Set Environment Variable option.

NOTE

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

The major sections in this chapter are:

- Current Directory
- Environment Macros
- Global Initialization File (mcutools.ini)
- Local Configuration File (usually project.ini)
- Paths
- Line Continuation
- Environment Variable Details

5.1 Current Directory

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

The current directory of a tool is determined by the operating system or by the program which launches another one.

- For the UNIX operating system, the current directory of an launched executable is also the current directory from where the binary file has been started.
- For MS Windows based operating systems, the current directory definition is 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 (e.g., an editor), the current directory is the one specified by the launching tool (e.g., current directory definition).
- For the tools, the current directory is where the local project file is located. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, use the environment variable DEFAULTDIR: Default Current Directory.

The current directory is displayed, with other information, using the -V: Prints the Compiler Version compiler option and in the *About* dialog box.

5.2 Environment Macros

You can use macros in your environment settings, as listed 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` is expanded to `C:\test\txt` and `OBJPATH` is expanded to `C:\test\obj`. You can use `$()` or `${}`. However, the referenced variable must be defined.

Special variables are also allowed (special variables are always surrounded by `{}` and they are case-sensitive). In addition, the variable content contains the directory separator `` `\' ``. The special variables are:

- `{Compiler}`

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

- `{Project}`

  Path of the current project file. This is used if the current project file is `C:\demo\project.ini`, and the variable contains `C:\demo\`.

- `{System}`

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

5.3 Global Initialization File (mcutools.ini)

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

Listing: Typical Global Initialization File Locations

```
C:\WINDOWS\mcutools.ini
CWInstallDir\MCU\prog\mcutools.ini
```

If a tool is started in the `CWInstallDir\<MCU>\prog` directory, the project file that is used is located in the same directory as the tool (`CWInstallDir\MCU\prog\mcutools.ini`).

If the tool is started outside the `CWInstallDir\MCU\prog` directory, the project file in the Windows directory is used (`C:\WINDOWS\mcutools.ini`).

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

5.4 Local Configuration File (usually project.ini)

All the configuration properties are stored in the configuration file. The same configuration file is used by different applications.

The shell uses the configuration file with the name `project.ini` in the current directory only. When the shell uses the same file as the compiler, the Editor Configuration is written and maintained by the shell and is 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. This plays a role when options, only available in one version, must be stored in the configuration file. In such situations, two files must be maintained for each different compiler version. If no incompatible options are enabled when the file is last saved, the same file may be used for both compiler versions.

The current directory is always the directory where the configuration file is located. If a configuration file in a different directory is loaded, the current directory also changes. When the current directory changes, the entire `default.env` file is reloaded. When a configuration file is loaded or stored, the options in the environment variable `COMPOPTIONS` are reloaded and added to the project options. This behavior is noticed when different `default.env` files exist in different directories, each containing incompatible options in the `COMPOPTIONS` variable.

When a project is loaded using the first `default.env`, its `COMPOPTIONS` are added to the configuration file. If this configuration is stored in a different directory where a `default.env` exists with incompatible options, the compiler adds options and remarks the inconsistency. You can remove the option from the configuration file with the option settings dialog box. You can also remove the option from the `default.env` with the shell or a text editor, depending which options are used in the future.

At startup, there are two ways to load a configuration:

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

If the `-Prod` option is used, the current directory is the directory the project file is in. If the `-Prod` option is used with a directory, the project.ini file in this directory is loaded.

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. Path names are declared using the following EBNF syntax.

Listing: EBNF path syntax

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

Most environment variables contain path lists directing where to look for files.

Listing: Environment variable path list with four possible paths.

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

If a directory name is preceded by an asterisk (`*`), the program recursively searches that entire directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list.

Listing: Setting an environment variable using recursive searching

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

NOTE

Some DOS environment variables (like `GENPATH`, `LIBPATH`, etc.) are used.

5.6 Line Continuation

It is possible to specify an environment variable in an environment file (`default.env`) over different lines using the line continuation character `\' (refer to the following listing).

Listing: Specifying an environment variable using line continuation characters

```
OPTIONS=\
      -W2 \

      -Wpd
```

This is the same as:

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

But this feature may not work well using it together with paths, e.g.:

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

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

This results in:

To avoid such problems, use a semicolon ';' at the end of a path if there is a `\' at the end (refer to the following listing):

Listing: Using a semicolon to allow a multiline 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	A synonym exists for some environment variables. Those 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 an 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 where possible. The example shows an entry in the default.env for a PC.
See also	Names related sections.

5.7.1 COMPOPTIONS: Default Compiler Options

Tools

Compiler

Synonym

HICOMPOPTIONS

Syntax

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

Arguments

`<option>`: Compiler command-line option

Default

None

Description

If this environment variable is set, the Compiler appends its contents to its command line each time a file is compiled. Use this variable to specify global options for every compilation. This frees you from having to specify them for every compilation.

NOTE

It is not recommended to use this environment variable if the Compiler used is version 5.x, because the Compiler adds the options specified in the `COMPOPTIONS` variable 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. This information is retrieved from the object files using the decoder.

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 will take the specified directory as their current directory instead of the one defined by the operating system or launching tool (e.g., editor).

NOTE

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

Specifying 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. Using `ENVIRONMENT` (e.g., set in the `autoexec.bat` (DOS) or `*.cshrc` (UNIX)), a different filename may be specified.

NOTE

This is a system level environment variable (global environment variable). It 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 is shown in the event of an improper error filename.

Examples

```
ERRORFILE=MyErrors.err
```

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

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

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

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

Lists 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. If you compile a file such as `sources\test.c`, an error list file, `\sources\test.err`, is generated.

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

For a source file such as `test.c`, an error list file with the name `\dir1\test.err` is generated.

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

For a source file such as `\dir1\dir2\test.c`, an error list file with the name `\dir1\dir2\errors.txt` is generated.

If the `ERRORFILE` environment variable is not set, the errors are written 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

If a header file is included with double quotes, the Compiler searches 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 whole directory tree is searched 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 data as strings. Whenever a new file is created by one of the tools, the new file gets a new time stamp entry.

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

The time stamp is retrieved from the object files using the decoder.

Example

```
INCLUDETIME=OFF
```

See also

Environment variables :

- COPYRIGHT: Copyright entry in object file
- USERNAME: User Name in Object File

5.7.8 LIBRARYPATH: `include <File>' Path

Tools

Compiler, ELF tools (Burner, Linker, or Decoder)

Synonym

LIBPATH

Syntax

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

Arguments

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

Default

Current directory

Description

If a header file is included with double quotes, the Compiler searches first in the current directory, then in the directories given by GENPATH: #include "File" Path and finally in the directories given by `LIBRARYPATH`.

NOTE

If a directory specification in this environment variable starts with an asterisk (`*`), the whole directory tree is searched 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

Environment variables :

- GENPATH: #include "File" Path
- USELIBPATH: Using LIBPATH Environment Variable
- Input Files

5.7.9 OBJPATH: Object File Path

Tools

Compiler, Linker, Decoder, Debugger, or Burner

Synonym

None

Syntax

```
OBJPATH=<path>
```

Default

Current directory

Arguments

`<path>`: Path without spaces

Description

If the Compiler generates an object file, the object file is placed into the directory specified by `OBJPATH`. If this environment variable is empty or does not exist, the object file is stored into the path where the source has been found.

If the Compiler tries to generate an object file specified in the path specified by this environment variable but fails (e.g., because the file is locked), the Compiler issues an error message.

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

Example

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

See also

Output Files

5.7.10 TEXTPATH: Text File Path

Tools

Compiler, Linker, or Decoder

Synonym

None

Syntax

```
TEXTPATH=<path>
```

Arguments

`<path>`: Path without spaces

Default

Current directory

Description

If the Compiler generates a textual file, the file is placed into the directory specified by `TEXTPATH`. If this environment variable is empty or does not exist, the text file is stored 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, or Librarian

Synonym

None

Syntax

```
TMP=<directory>
```

Arguments

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

Default

None

Description

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

NOTE

This is an environment variable on 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 environment variable `LIBRARYPATH` is used by the Compiler to look for system header files `<*.h>`.

`NO, OFF`: The environment variable `LIBRARYPATH` is not used by the Compiler.

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 (e.g., 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, or, Librarian

Synonym

None

Syntax

```
USERNAME=<user>
```

Arguments

`<user>`: Name of user

Default

None

Description

Each object file contains an entry identifying the user who created the object file. This information is retrievable from the object files 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 input and output files and file processing.

- Input Files
- Output Files
- File Processing

6.1 Input Files

The following input files are described:

- Source Files
- Include Files

6.1.1 Source Files

The 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 have the `*.c` extension and that all header files use the `*.h` extension. Source files are searched first in the Current Directory and then in the GENPATH: #include "File" Path directory.

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 that are included using double quotes as in:

```
#include "test.h"
```

are searched first in the current directory, then in the directory specified by the -I: Include File Path option, then in the directories given in the GENPATH: #include "File" Path environment variable, and finally in those listed in the `LIBPATH` or `LIBRARYPATH`: `` `include <File>' Path `` environment variable. The current directory is set using the IDE, the Program Manager, or the DEFAULTDIR: Default Current Directory environment variable.

Include files that are included using angular brackets as in

```
#include <stdio.h>
```

are searched for first in the current directory, then in the directory specified by the -I option, and then in the directories given in `LIBPATH` or `LIBRARYPATH`. The current directory is set using the IDE, the Program Manager, or the `DEFAULTDIR` environment variable.

6.2 Output Files

The following output files are described:

- 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. This file is written to the directory listed in the OBJPATH: Object File Path environment variable. If that variable contains more than one path, the object file is written in the first listed directory. If this variable is not set, the object file is written in the directory the source file was found. Object files always get the extension `*.o`.

6.2.2 Error Listing

If the Compiler detects any errors, it does not create an object file. Rather, it creates an error listing file named `err.txt`. This file is generated in the directory where the source file was found (also see ERRORFILE: Error filename Specification).

If the Compiler's window is open, it displays the full path of all header files read. After successful compilation the number of code bytes generated and the number of global objects written to the object file are also displayed.

If the Compiler is started from an IDE (with `'%f'` given on the command line) or CodeWright (with `'%b%e'` given on the command line), this error file is not produced. Instead, it writes the error messages in a special format in a file called `EDOUT` using the Microsoft format by default. You may use the CodeWrights's *Find Next Error* command to display both the error positions and the error messages.

6.2.2.1 Interactive Mode (Compiler Window Open)

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

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

6.2.2.2 Batch Mode (Compiler Window not Open)

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

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

6.3 File Processing

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

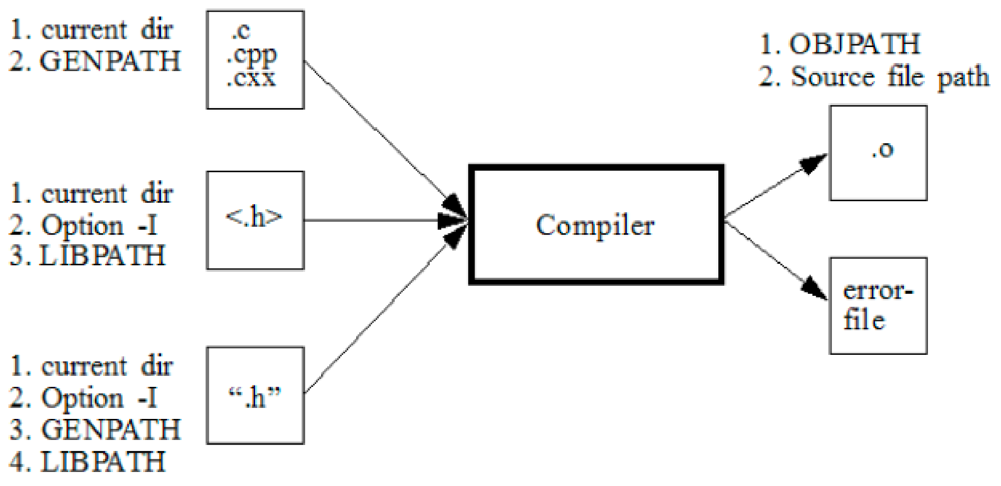


Figure 6-1. Files used with the Compiler

Chapter 7
Compiler Options

The Compiler provides a number of Compiler options that control the Compiler's operation. Options consist of a minus sign or dash (`-`), followed by one or more letters or digits. Anything not starting with a dash or minus sign is the name of a source file to be compiled. You can specify Compiler options on the command line or in the `COMPOPTIONS` variable. Each Compiler option is specified only once per compilation.

Command line options are not case-sensitive, e.g., `-Li` is the same as `-li`.

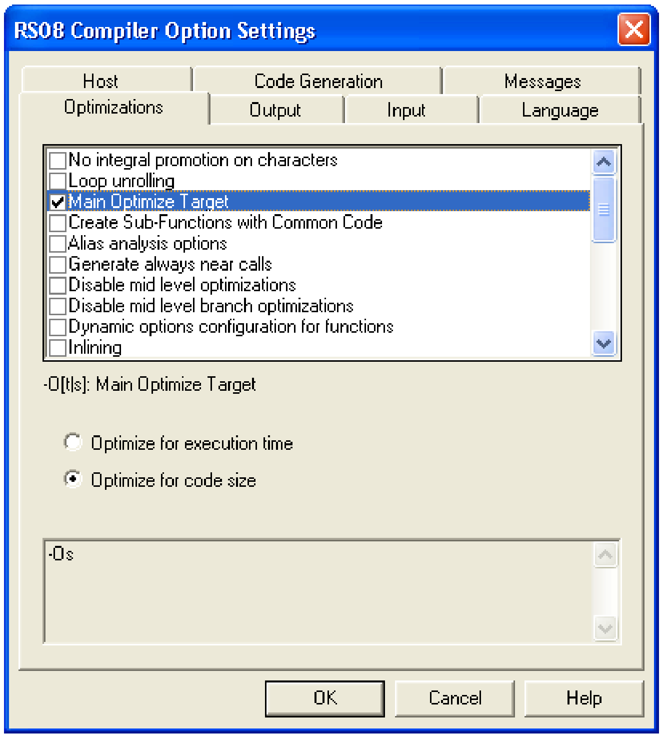
NOTE

It is not possible to coalesce options in different groups, e.g., - `Cc` - `Li`*cannot* be abbreviated by the terms - `Cci` or - `CcLi`.

Another way to set the compiler options is to use the RS08 Compiler Option Settings dialog box.

NOTE

Do not use the `COMPOPTIONS` environment variable if the GUI is used. The Compiler stores the options in the `project.ini` file, not in the `default.env` file.



RS08 Compiler Option Settings

Tabs: Host | Code Generation | Messages | Optimizations | Output | Input | Language

- [ ] No integral promotion on characters
- [ ] Loop unrolling
- [✓] Main Optimize Target
- [ ] Create Sub-Functions with Common Code
- [ ] Alias analysis options
- [ ] Generate always near calls
- [ ] Disable mid level optimizations
- [ ] Disable mid level branch optimizations
- [ ] Dynamic options configuration for functions
- [ ] Inlining

-O[t|s]: Main Optimize Target

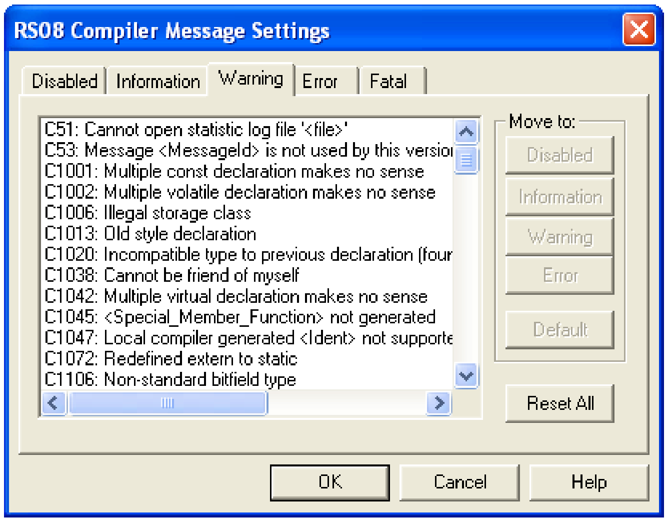
○ Optimize for execution time

● Optimize for code size

-Os

Figure 7-1. Option Settings Dialog Box

The **RS08 Compiler Message Settings** dialog box, shown in the following figure, may also be used to move messages (`-Wmsg` options).



RS08 Compiler Message Settings

Tabs: Disabled | Information | Warning | Error | Fatal

```
C51: Cannot open statistic log file '<file>'
C53: Message <MessageId> is not used by this versio
C1001: Multiple const declaration makes no sense
C1002: Multiple volatile declaration makes no sense
C1006: Illegal storage class
C1013: Old style declaration
C1020: Incompatible type to previous declaration (four
C1038: Cannot be friend of myself
C1042: Multiple virtual declaration makes no sense
C1045: <Special_Member_Function> not generated
C1047: Local compiler generated <Ident> not supporte
C1072: Redefined extern to static
C1106: Non-standard bitfield type
```

Figure 7-2. RS08 Compiler Message Settings Dialog Box

The major sections of this chapter are:

- Option Recommendation : Advice about the available compiler options.
- Compiler Option Details : Description of the layout and format of the compiler command-line options that are covered in the remainder of the chapter.

7.1 Option Recommendation

Depending on the compiled sources, each Compiler optimization may have its advantages or disadvantages. The following are recommended:

- -Wpd: Error for Implicit Parameter Declaration

The default configuration enables most optimizations in the Compiler. If they cause problems in your code (e.g., they make the code hard to debug), switch them off (these options usually have the -On prefix). Candidates for such optimizations are peephole optimizations.

Some optimizations may produce more code for some functions than for others (e.g., -Oi: Inlining or -Cu: Loop Unrolling). Try those options to get the best result for each.

To acquire the best results for each function, compile each module with the -OdocF: Dynamic Option Configuration for Functions option. An example for 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

NOTE

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

This topic describes:

- 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

A special group is the STARTUP group: The options in this group cannot be specified interactively; they can only be specified on the command line to start the tool.

Refer to the following table:

Table 7-1. Compiler Option Groups

Group	Description
`HOST`	Lists options related to the host
`LANGUAGE`	Lists options related to the programming language (e.g., ANSI-C)
`OPTIMIZATIONS`	Lists optimization options
`OUTPUT`	Lists output file generation options (types of file generated)
`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`	Options specified only on tool startup

The 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 (e.g., the option to set the type sizes).

7.2.2 Option Scopes

Each option has also a scope. Refer to the following table.

Table 7-2. Option Scopes

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

The available options are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheets.

7.2.3 Option Detail Description

The remainder of this section describes each of the Compiler options available for the Compiler. The options are listed in alphabetical order. Each is divided into several sections listed in the following table.

Table 7-3. Compiler Option - Documentation Topics

Topic	Description
Group	HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGES, or VARIOUS.
Scope	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	List of defines related to the compiler option
Pragma	List of pragmas related to the compiler option
Description	Provides a detailed description of the option and how to use it
Example	Gives an example of usage, and effects of the option where possible. compiler settings, source code and Linker PRM files are displayed where 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

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

The following 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)`	Replaces it with the contents of an environment variable
`%%`	Generates a single `%'

7.2.3.1.1 Examples

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

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

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

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

```
TheWhole
```

`%n` returns just the filename without extension:

```
TheWholeThing
```

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

```
myE
```

`%e` is used for the whole extension:

```
myExt
```

`%f` gives the path plus the filename:

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

Because the path contains a space, using `%"` or `%'` is recommended: Thus, `%"%f%"` results in: (using double quotes)

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

where `%'%f%'` results in: (using single quotes)

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

`%(`*envVariable*`)` uses an environment variable. A file separator following after `%(`*envVariable*`)` is ignored if the environment variable is empty or does not exist. In other words, if `TEXTPATH` is set to: `TEXTPATH=C:\Freescale\txt`, `%(TEXTPATH)\myfile.txt` is replaced with:

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

But if `TEXTPATH` does not exist or is empty, `%(TEXTPATH)\myfile.txt` is set to:

```
myfile.txt
```

A `%%` may be used to print a percent sign. Using `%e%%` results in:

```
myExt%
```

7.2.3.2 -!: Filenames are Clipped to DOS Length

Group

INPUT

Scope

Compilation Unit

Syntax

```
-!
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option, called *cut*, is very useful when compiling files copied from an MS-DOS file system. Filenames are clipped to DOS length (eight characters).

Listing: Example of the cut option, -!

```
The cut option truncates the following include directive: #include "mylongfilename.h"
to:
#include "mylongfi.h"
```

7.2.3.3 -AddIncl: Additional Include File

Group

INPUT

Scope

Compilation Unit

Syntax

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

Arguments

`<fileName>`: name of the included file

Default

None

Defines

None

Pragmas

None

Description

Includes the specified file at the beginning of the compilation unit. It has the same effect as if written at the beginning of the compilation unit using double quotes (`"..."`):

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

Example

See the following listing for the `-AddIncl` compiler option that includes the above header file.

Listing: -AddIncl example

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

See also

-I: Include File Path compiler option

7.2.3.4 -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 `-Ansi` is specified, all non ANSI-compliant keywords (e.g., `__asm`, `__far` and `__near`) are not accepted by the Compiler, and the Compiler 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, the -Cppc: C++ Comments in ANSI-C compiler option must be set.

The `asm` keyword is also not allowed if `-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.

7.2.3.5 -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 crs08.exe -ArgFileoption.txt command line will be equivalent to crs08.exe -Mt.

7.2.3.6 -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

```
-BfaBLS
```

Defines

`__BITFIELD_MSWORD_FIRST__`

`__BITFIELD_LSWORD_FIRST__`

`__BITFIELD_MSBYTE_FIRST__`

`__BITFIELD_LSBYTE_FIRST__`

`__BITFIELD_MSBIT_FIRST__`

`__BITFIELD_LSBIT_FIRST__`

Pragmas

None

Description

Normally, bits in byte bitfields are allocated from the least significant bit to the most significant bit. This produces less code overhead if a byte bitfield is allocated only partially.

Example

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

Listing: Example struct used for the next listing

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

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

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

Listing: Examples of changing the bitfield allocation order

```
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

7.2.3.7 -BfaGapLimitBits: Bitfield Gap Limit

Group

CODE GENERATION

Scope

Function

Syntax

```
-BfaGapLimitBits<number>
```

Arguments

`<number>`: positive number specifying the maximum number of bits for a gap

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 in the following listing, assume that you have specified a 3-bit maximum gap, i.e., `-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 zero, 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 the 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 m

The following listing specifies a maximum size of two bits for a gap.

Listing: Example where the maximum number of gap bits is two

```
-BfaGapLimitBits2
```

See also

Bitfield Allocation

7.2.3.8 -BfaTSR: Bitfield Type-Size Reduction

Group

CODE GENERATION

Scope

Function

Syntax

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

Arguments

`ON`: Bitfield Type Size Reduction enabled

`OFF`: Bitfield Type-Size Reduction disabled

Default

```
-BfaTSRON
```

Defines

`__BITFIELD_TYPE_SIZE_REDUCTION__`

`__BITFIELD_NO_TYPE_SIZE_REDUCTION__`

Pragmas

None

Description

This option is configurable whether or not the compiler uses type-size reduction for bitfields. Type-size reduction means that the compiler can reduce the type of an `int` bitfield to a `char` bitfield if it fits into a character. This 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

7.2.3.9 -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 things 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 were set.

Example

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

See Also

-Cn: Disable compactC++ features

7.2.3.10 -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 Linker prepares no initialization for 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 (see the *Linker* manual).

With #pragma `CONST_SECTION` for constant segment allocation, variables declared as `const` are allocated in this segment.

If the current data segment is not the default segment, `const` objects in that user-defined segment are not allocated in the `ROM_VAR` segment but remain in the segment defined by the user. If that data segment happens to contain *only*`const` objects, it may be allocated in a ROM memory section (for more information, refer to the *Linker* section of the Build Tools manual).

NOTE

In the ELF/DWARF object-file format, constants are allocated into the `.rodata` section.

NOTE

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

Example

See also

Segmentation

Linker section in the Build Tools manual

-F (-F2, -F2o): Object File Format option

#pragma INTO_ROM: Put Next Variable Definition into ROM

7.2.3.11 -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 Cosmic style `@near`, `@far` and `@tiny` space modifiers as well as `@interrupt`in your C code. The `-ANSI` option must be switched off. It is not necessary to remove the Cosmic space modifiers from your application code. There is no need to place the objects to sections addressable by the Cosmic space modifiers.

The following is done when a Cosmic modifier is parsed:

- The objects declared with the space modifier are always allocated in a special Cosmic compatibility (`_CX`) section (regardless of which section pragma is set) depending on the space modifier, on the `const` qualifier or if it is a function or a variable.
- Space modifiers on the left hand side of a pointer declaration specify the pointer type and `pointer size`, depending on the target.

See the example in the **Listing: Cosmic Space Modifiers** for a `prm` file describing the placement of sections mentioned in the following table.

Table 7-5. 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 the CodeWarrior IDE, refer to the technical note *TN234*. The following table indicates how space modifiers are mapped for the RS08.

Table 7-6. Cosmic Space modifier mapping for the RS08

Definition	Keyword Mapping
`@tiny`	`ignored`
`@near`	`ignored`
`@far`	`ignored`

Refer to 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, Linker Manual, TN234

7.2.3.12 -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, they are replaced with keywords as shown in the following table.

Table 7-7. 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.

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;

    }

  }

}
```

7.2.3.13 -Cn: Disable compactC++ features

Group

LANGUAGE

Scope

Compilation Unit

Syntax

```
-Cn [= {Vf|Tpl|Ptm|Mih|Ctr|Cpr}]
```

Arguments

`Vf`: Do not allow virtual functions

`Tpl`: Do not allow templates

`Ptm`: Do not allow pointer to member

`Mih`: Do not allow multiple inheritance and virtual base classes

`Ctr`: Do not create compiler defined functions

`Cpr`: Do not allow class parameters and class returns

Default

None

Defines

None

Pragmas

None

Description

If the `-C++c` option is enabled, you can disable the following compactC++ features:

- `Vf` : Virtual functions are not allowed.

  Avoid having virtual tables that consume a lot of memory.
- `Tpl` : Templates are not allowed.

  Avoid having many generated functions perform similar operations.
- `Ptm` : Pointer to member not allowed.

  Avoid having pointer-to-member objects that consume a lot of memory.
- `Mih` : Multiple inheritance is not allowed.

  Avoid having complex class hierarchies. Because virtual base classes are logical only when used with multiple inheritance, they are also not allowed.
- `Ctr` : The C++ Compiler can generate several kinds of functions, if necessary:
  - Default Constructor

- Copy Constructor
- Destructor
- Assignment operator

  With this option enabled, the Compiler does not create those functions. This is useful when compiling C sources with the C++ Compiler, assuming you do not want C structures to acquire member functions.

- `Cpr` : Class parameters and class returns are not allowed.

  Avoid overhead with Copy Constructor and Destructor calls when passing parameters, and passing return values of class type.

Example

```
  -C++c -Cn=Ctr
```

7.2.3.14 -Cni: No Integral Promotion

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Cni
```

Arguments

None

Default

None

Defines

`__CNI__`

Pragmas

None

Description

Enhances code density of character operations by omitting integral promotion. This option enables a non ANSI-C compliant behavior.

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 results in a zero-extension of each character operand, and then adding them back in as int operands. If the result must be stored back into a character, this integral promotion is not necessary. When this option is set, promotion is avoided where possible.

The code size may be decreased if this option is set because operations may be performed on a character base instead of an integer base.

The `-Cni` option enhances character operation code density by omitting integral promotion.

Consider the following:

- In most expressions, ANSI-C requires char type variables to be extended to the next larger type int, which is required to be at least 16-bit in size by the ANSI standard.
- The `-Cni` option suppresses this ANSI-C behavior and thus allows 'characters' and 'character sized constants' to be used in expressions. This option does not conform to ANSI standards. Code compiled with this option is not portable.
- The ANSI standard requires that 'old style declarations' of functions using the `char` parameter, as shown in the below listing, be extended to `int`. The `-Cni` option disables this extension and saves additional RAM.

Example

See the following listing for an example of "no integer promotion."

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 just 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, (i.e., 128, 129, etc.) are treated as integer. If you want them treated as characters, they must be cast (refer to the following listing).
  Listing: Casting integers to signed char

```
signed char a, b;
if (a > b * (signed char)129)
```

NOTE

This option is ignored with the `-Ansi` Compiler switch active.

NOTE

With this option set, the code that is generated does not conform to the ANSI standard. In other words: the code generated is wrong if you apply the ANSI standard as reference. Using this option is not recommended in most cases.

7.2.3.15 -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 //), this option may be used to allow C++ comments (refer to 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 foo(void) // this is a C++ comment
```

See also

-Ansi: Strict ANSI

7.2.3.16 -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 its members are `constant` or `volatile`, respectively. Consider the following example.

Example

The source code in the following listing declares two structs, each of which has a `const` member.

Listing: Be careful to not write to a constant struct

```
struct {
  const field;

} s1, s2;

void foo(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.

7.2.3.17 -CswMaxLF: Maximum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

```
-CswMaxLF<number>
```

Arguments

`<number>`: a number in the range of 0 - 100 denoting the maximum load factor

Default

Backend-dependent

Defines

None

Pragmas

None

Description

Allows changing 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 if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an `unrolled' binary search in a table which 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 are generated in specific `fuel' status:

The table in the following listing is filled to 90% (labels for `0' to `9', except for `5').

Listing: Load factor example

```
switch(i) {
 case 0: ...
 case 1: ...
 case 2: ...
 case 3: ...
 case 4: ...
// case 5: ...
 case 6: ...
 case 7: ...
 case 8: ...
 case 9: ...
 default
}
```

Assumed that the minimum load factor is set to 50% and setting the maximum load factor for the above case to 80%, a branch tree is generated instead a table. But setting the maximum load factor to 95% produces a table.

To guarantee that tables are generated for switches with full 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
- -CswMinSLB: Minimum Number of Labels for Switch Search Tables
- -CswMinLF: Minimum Load Factor for Switch Tables

7.2.3.18 -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

This option allows changing the default strategy of the Compiler using 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 8 labels (case entries) (actually this number is highly backend-dependent). If there are not enough labels for a table, a branch tree is generated (tree of if-else-if-else). This branch tree is like an `unrolled' binary search in a table which evaluates very fast the associated label for a switch expression.

Using a branch tree instead of a table may increases the code execution speed, but it probably increases the code size. 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, just set the minimum number of labels needed for a table to a high value (e.g., 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

7.2.3.19 -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 8 labels and if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an `unrolled' binary search in a table which 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 are generated in specific `fuel' status:

The table in the following listing is filled to 90% (labels for `0' to `9', except for `5').

Listing: Load factor example

```
switch(i) {
 case 0: ...
 case 1: ...
 case 2: ...
 case 3: ...
 case 4: ...
 // case 5: ...
 case 6: ...
 case 7: ...
 case 8: ...
 case 9: ...
 default
}
```

Assuming that the maximum load factor is set to 100% and the minimum load factor for the above case is set to 90%, this still generates a table. But setting the minimum load factor to 95% produces a branch tree.

To guarantee that tables are generated for switches with full 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

7.2.3.20 -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.

Switch tables are implemented in different ways. When almost all case entries in some range are given, a table containing only branch targets is used. Using such a dense table is efficient because only the correct entry is accessed. When large holes exist in some areas, a table form can still be used.

But now the case entry and its corresponding branch target are encoded in the table. This is called a search table. A complex runtime routine must be used to access a search table. This routine checks 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 time an entry in a search table must be found, all previous entries must be checked first. For a dense table, the right offset is computed and accessed. In addition, note that all backends implement search tables (if at all) by using a complex runtime routine. This may make debugging more complex.

To disable search tables for switch statements, set the minimum number of labels needed for a table to a high value (e.g., 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

7.2.3.21 -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, e.g.,

  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;
}
```

When the `-Cu` compiler option is used, the Compiler issues an information message *Unrolling loop* and transforms this loop as shown in the following listing:

Listing: Transformation of the 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 a 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 a loop with only one pass

```
for (i=1; i<2; i++) {   j += i;
}
```

The Compiler issues a warning '*Unrolling loop*' and transforms the `for` loop into

```
j += 1;
i  = 2;
```

because the loop body is executed only once.

7.2.3.22 -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. No object code is generated, though the Compiler performs a syntactical check of the source code. This allows a quick test if the Compiler accepts the source without errors.

7.2.3.23 -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 `<a 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.

```
-DDEBUG=0
```

This is the same as writing:

```
#define DEBUG 0
```

in the source file.

If you need strings with blanks in your macro definition, there are two ways. Either use escape sequences or 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.

7.2.3.24 -Ec: Conversion from 'const T*' to 'T*'

Group

LANGUAGE

Scope

Function

Syntax

```
-Ec
```

Arguments

None

Default

None

Description

If this non-ANSI compliant extension is enabled, a pointer to a constant type is treated 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 if some older source has to be compiled.

Defines

None

Pragmas

None

Examples

See the following listings for examples using `-Ec` conversions.

Listing: Conversion from 'const T*' to 'T*

```
void f() {
  int *i;

  const int *j;

  i=j; /* C++ illegal, but OK with -Ec! */

}

struct A {

  int i;

};

void g() {

  const struct A *a;

  a->i=3; /* ANSI C/C++ illegal, but OK with -Ec! */

}

void h() {

  const int *i;

  *i=23; /* ANSI-C/C++ illegal, but OK with -Ec! */

}
```

Listing: Assigning a value to a "constant" pointer

```
-Ec
void foo(const int *p){

  *p = 0; // Some Compilers do not issue an error.
```

7.2.3.25 -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 (for more information, refer to the Using Special Modifiers topic).

Default

The default filename is `%f.e%e`. A file named `fun.c` creates an encrypted file named `fun.ec`.

Description

All files passed together with this option are encrypted using the given key with the -Ekey: Encryption Key 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

```
fun.c fun.h -Ekey1234567 -Eencrypt=%n.e%e
```

This encrypts the `fun.c` file using the `1234567` key to the `fun.ec` file, and the `fun.h` file to the `fun.eh` file.

The encrypted `fun.ec` and `fun.eh` files may be passed to a client. The client is able to compile the encrypted files without the key by compiling the following file:

```
fun.ec
```

See also

-Ekey: Encryption Key

7.2.3.26 -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

This option is used 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

```
fun.c -Ekey1234567 -Eencrypt=%n.e%e
```

This encrypts the `fun.`' file using the `1234567` key.

See also

-Eencrypt: Encrypt Files

7.2.3.27 -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. This environment variable may be used in the maker, or used 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 using filenames with spaces:

```
  -Env"OBJPATH=\program files"
```

See also

Environment

7.2.3.28 -F (-F2, -F2o): Object File Format

Group

OUTPUT

Scope

Application

Syntax

```
  -F(2|2o)
```

Arguments

`2`: ELF/DWARF 2.0 object-file format

`2o`: compatible ELF/DWARF 2.0 object-file format

NOTE

Not all object-file formats may be available for a target.

Default

`-F2`

Defines

`__ELF_OBJECT_FILE_FORMAT__`

Pragmas

None

Description

The Compiler writes the code and debugging info after compilation into an object file. The Compiler produces an ELF/DWARF object file when the `-F2` option is 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 were not conforming 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 new 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. If you want to use the older versions, use `-f2o` instead of `-f2`. New versions of the debugger are able to load both formats correctly. Also, some older ELF/DWARF object file loaders from emulator vendors may require you to set the `-F2o` option.

Note that it is recommended to 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 (e.g., floating point format). Debug information is also more robust.

7.2.3.29 -Fd: Double is IEEE32

Group

CODE GENERATION

Scope

Application

Syntax

```
-Fd
```

Arguments

None

Default

None

Defines

See -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.

7.2.3.30 -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 list (i.e., 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 you invoke the `-H` option.

Example

The following listing lists 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
```

7.2.3.31 -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: `include <File>' Path and GENPATH: #include "File" Path environment variables. Paths specified with this option have precedence over includes in the current directory, and 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

7.2.3.32 -IEA: Specify the address of the interrupt exit address register

Specifies the address of the interrupt exit address register. By default, it is `0x200`.

Group

CODE GENERATION

Scope

Compilation Unit

7.2.3.33 -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 (see 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. The name of the created file is specified by this option. If no name is specified, a default of `%f.inc` is taken. 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 `%f` is used, the generated file is 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

-La: Generate Assembler Include File

7.2.3.34 -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 (see 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. All assembler generated instructions are also printed to this file. The name of the file is specified by this option. If no name is specified, a default of `%n.lst` is taken. The TEXTPATH: Text File Path environment variable is used if the resulting filename contains no path information.

The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, this option can only be used to review the generated code. It can not currently be used to generate a file for assembly.

Example

```
-Lasm=asm.lst
```

See also

7.2.3.35 -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

`c`: Do not write the code

`i`: Do not write the instruction

`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 has the layout shown in the following listing. The letters in brackets ([]) indicate which suboption may be used to switch it off:

Listing: -Lasm configuration options

```
[v] ANSI-C/cC++ Compiler V-5.0.1
[v]

[p]     1:

[p]     2:   void foo(void) {

[h]

[h] Function: foo

[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
```

7.2.3.36 -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 is `predef.h`, but may be changed (e.g., `-Ldf="myfile.h"`). The file is generated in the directory specified by the TEXTPATH: Text File Path environment variable. The defines written to this file depend on the actual Compiler option settings (e.g., 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` which was used to compile all sources.

NOTE

This option only has an effect if a file is compiled. This option is unusable if you are not compiling a file.

Example

The following listing is an example which lists the contents of a file containing define directives.

Listing: Displays the contents of a file where define directives are present

```
-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

7.2.3.37 -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 files are stored 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 is an example where the `-Li` compiler option can be used to display a file's contents when that file contains an included directive.

Listing: Display contents of a file when include directives are present

```
If the source file is:
 D:\Compile\Sources\main.c:
/*D:\Compile\Sources\main.c*/
#include <hidef.h>
#include "derivative.h"
Then the generated file is:

D:\Compile\Sources\main.c : \

"C:\Freescale\CW MCU V10.x\MCU\lib\hc08c\include\hidef.h" \
"C:\Freescale\CW MCU V10.x\MCU\lib\hc08c\include\stddef.h" \
"C:\Freescale\CW MCU V10.x\MCU\lib\hc08c\include\stdtypes.h" \
D:\Compile\Sources\derivative.h \
D:\Compile\Sources\MC9S08GT32.h
```

See also

-Lm: List of Included Files in Make Format

7.2.3.38 -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 (e.g., if it is a demo version or a full version). This information is also displayed in the about box.

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 License Server

7.2.3.39 -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 (e.g., if the tool or feature is a demo version or a full version) of every tool or `*.dll` in the directory where the executable is located. Each file in the directory is analyzed.

Example

```
-LicA
```

See also

Compiler options:

- -Lic: License Information
- -LicBorrow: Borrow License Feature
- -LicWait: Wait until Floating License is Available from Floating License Server

7.2.3.40 -LicBorrow: Borrow License Feature

Group

HOST

Scope

None

Syntax

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

Arguments

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

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

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

Default

None

Defines

None

Pragmas

None

Description

This option allows 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.

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

You can check the status of currently borrowed features in the tool about box.

NOTE

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

Example

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

See also

Compiler options:

- -LicA: License Information about every Feature in Directory
- -Lic: License Information
- -LicWait: Wait until Floating License is Available from Floating License Server

7.2.3.41 -LicWait: Wait until Floating License is Available from Floating License 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 will immediately return. With `-LicWait` set, the application will wait (blocking) until a license is available from the floating license server.

Example

```
-LicWait
```

See also

- -Lic: License Information
- -LicA: License Information about every Feature in Directory
- -LicBorrow: Borrow License Feature

7.2.3.42 -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

The `-Ll` option causes the Compiler to append statistical information about the compilation session to the specified file. Compiler options, code size (in bytes), memory usage (in bytes) and compilation time (in seconds) are given for each procedure of the compiled file. The information is appended to the specified filename (or the file `make.txt`, if no argument given). If the TEXTPATH: Text File Path environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

Example

The following listing is an example where the use of the `-Ll` compiler options allows statistical information to be added to the end of an output listing file.

Listing: Statistical information appended to an assembler listing

```
-Ll=mylog.txt
/* fun.c */
int Func1(int b) {
 int a = b+3;
 return a+2;
}
void Func2(void) {
}
```

Appends the following two lines into `mylog.txt`:

```
fun.c Func1 -Ll=mylog.txt    11   4 0.055000

fun.c Func2 -Ll=mylog.txt         1   0 0.001000
```

7.2.3.43 -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

The `-Lm` option causes the Compiler to generate a text file which contains a list of the `#include` files specified in the source. The generated list is in a make format. The `-Lm` option is useful 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 information is appended to the specified filename (or the `make.txt` file, if no argument is given). If the TEXTPATH: Text File Path environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

Example

The following listing is an example where the `-Lm` option generates a make file containing include directives.

Listing: Make file construction

```
COMPOTIONS=-Lm=mymake.txt Compiling the following sources 'foo.c' and 'second.c':
/* foo.c */
#include <stddef.h>
#include "myheader.h"
...
/* second.c */
#include "inc.h"
#include "header.h"
...
This adds the following entries in the 'mymake.txt':
foo.o : foo.c stddef.h myheader.h
second.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>])

7.2.3.44 -LmCfg: Configuration for List of Included Files in Make Format (option -Lm)

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-LmCfg[={i|l|m|n|o|u|q}]
```

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

This option is used when configuring the -Lm: List of Included Files in Make Format option. The `-LmCfg` option is operative only if the `-Lm` option is also used. The `-Lm` option produces the `dependency' information for a make file. Each dependency information grouping is structured as shown in the following listing:

Listing: Dependency information grouping

```
<main object file>: <main source file> {<included file>}
```

Example

If you compile a file named `b.c`, which includes `` `stdio.h' ``, the output of `-Lm` may be:

```
b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The `l` suboption uses line continuation for each single entry in the dependency list. This improves readability as shown in the following listing:

Listing: l suboption

```
b.o:    \   b.c    \
  stdio.h \
  stddef.h \
  stdarg.h \
  string.h
```

With the `m` suboption, the full path of the main file is written. The main file is the actual compilation unit (file to be compiled). This is necessary if 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 `o` suboption has the same effect as `m`, but 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 `i` suboption writes the full path of all included files in the dependency list (refer to the following listing):

Listing: i suboption

```
main.o: main.c "C:\Freescale\CW MCU V10.x\MCU\lib\hc08c\include\hidef.h" "C:\Freescale\CW
MCU V10.x\MCU\lib\hc08c\include\stddef.h" "C:\Freescale\CW MCU V10.x\MCU\lib\hc08c\include
\stdtypes.h"
D:\Compile\Sources\derivative.h D:\Compile\Sources\MC9S08GT32.h
```

The `u` suboption updates the information in the output file. If the file does not exist, the file is created. If the file exists and the current information is not yet in the file, the information is appended to the file. If the information is already present, it is updated. This allows you to specify this suboption for each compilation ensuring that the make dependency file is always up to date.

Example

```
COMPOTIONS=-LmCfg=u
```

See also

Compiler options :

- -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

7.2.3.45 -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

The `-Lo` option causes the Compiler to append the object filename to the list in the specified file.The information is appended to the specified filename (or the file `make.txt` file, if no argument given). If the TEXTPATH: Text File Path is set, the file is stored into the path specified by the environment variable. Otherwise, it is stored in the current directory.

See also

Compiler options:

- -Li: List of Included Files to ".inc" File
- -Lm: List of Included Files in Make Format

7.2.3.46 -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 (see Using Special Modifiers).

Default

No file created

Defines

None

Pragmas

None

Description

The -Lp option causes the Compiler to generate a text file which contains 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 `TEXTPATH` environment variable is used to store the preprocessor file.

The resultant file is a form of the source file. All preprocessor commands (i.e., `#include`, `#define`, `#ifdef`, etc.) have been resolved. Only source code is listed with line numbers.

See also

-LpX: Stop after Preprocessor

-LpCfg: Preprocessor Output Configuration

7.2.3.47 -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 how source file and -line information is formatted in the preprocessor output. Switching `-LpCfg` off means that the output is formatted as in former compiler versions. The effects of the arguments are listed in the following table.

Table 7-8. 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 unsigned int size_t ;/* 35 */ typedef signed int ptrdiff_t ;`
`e`	`int j;int i;`	`int j;int i;`
`f`	`/**** FILE '`*<CWInstallDir>*`\MCU \lib\hc08c\include\hidef.'*/`	`/**** FILE 'hidef.h' */`
`l`	`#line 1 "hidef.h"`	`/**** FILE 'hidef.h' */`
n	/* 9 */ foo ("abc" "def") ;	/* 9 */ foo ("abcdef") ;

Table continues on the next page...

Table 7-8. Effects of Source and Line Information Format Control Arguments (continued)

Argument	on	off
`m`		`/**** FILE 'hidef.h' */`
`s`	`/* 22 */ typedef unsigned int size_t;/* 35 */ typedef signed int ptrdiff_t;/* 44 */ typedef unsigned char wchar_t;`	`/* 22 */ typedef unsigned int size_t ;/* 35 */ typedef signed int ptrdiff_t ;/* 44 */ typedef unsigned char 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

7.2.3.48 -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

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

Group

CODE GENERATION

Scope

Application

Syntax

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

Arguments

`b`: BANKED memory model

`s`: SMALL memory model

Default

```
-Ms
```

Defines

`__BANKED__`

`__SMALL__`

Pragmas

None

Description

See Memory Models for details.

Example

```
-Ms
```

7.2.3.50 -N: Show Notification Box in Case of Errors

Group

MESSAGES

Scope

Function

Syntax

`-N`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Makes the Compiler display an alert box if there was an error during compilation. This is useful when running a make file (see *Make Utility*) because the Compiler waits for you to acknowledge the message, thus suspending make file processing. The `N` stands for "Notify".

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

Example

```
-N
```

If an error occurs during compilation, a dialog box appears.

7.2.3.51 -NoBeep: No Beep in Case of an Error

Group

MESSAGES

Scope

Function

Syntax

```
-NoBeep
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

There is a beep notification at the end of processing if an error was generated. To implement a silent error, this beep may be switched off using this option.

Example

```
-NoBeep
```

7.2.3.52 -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 RS08 has a read cycle. This may lead to unwanted lateral effects (e.g. if the variable is mapped over a hardware register).

7.2.3.53 -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. When this option is used, the compiler does not generate 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. This compiler option has to be used 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 (-F2, -F2o): Object File Format
- -NoPath: Strip Path Info

7.2.3.54 -NoPath: Strip Path Info

Group

OUTPUT

Scope

Compilation Unit

Syntax

```
-NoPath
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option set, it is possible to avoid any path information in object files. This is useful if you want to move object files to another file location, or to hide your path structure.

See also

-NoDebugInfo: Do Not Generate Debug Information

7.2.3.55 -Oa: Alias Analysis Options

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Oa (addr|ANSI|type|none)
```

Arguments

`addr`: All objects in same address area may overlap (safe mode, default)

`ANSI`: use ANSI99 rules

`type`: only objects in same address area with same type may overlap

`none`: assume no objects do overlap

Default

addr

Defines

None

Pragmas

None

Description

These four different options allow the programmer to control the alias behavior of the compiler. The option `-oaaddr` is the default because it is safe for all C programs. Use option `-oaansi` if the source code follows the ANSI C99 alias rules. If objects with different types never overlap in your program, use option `-oatype`. If your program doesn't have aliases at all, use option `-oanone` (or the ICG option `-ona`, which is supported for compatibility reasons).

Examples

```
-oaANSI
```

7.2.3.56 -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

There are vario us points where the Compiler has to choose between two possibilities: it can either generate fast, but large code, or small but slower code.

The Compiler generally optimizes on code size. It often has to decide between a runtime routine or an expanded code. The programmer can decide whether to choose between 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 replaces slower/smaller code with faster/larger code.

NOTE

This option only affects some special code sequences. This option has to be set together with other optimization options (e.g., register optimization) to get best results.

Example

```
-Os
```

7.2.3.57 -O[nf|f]: Create Sub-Functions with Common Code

Group

OPTIMIZATIONS

Description

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 of functions having arguments on the stack.

7.2.3.58 -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 has the same name as the processed source file, but with the `*.o` extension. This option allows a flexible way to define the object filename. It may contain special modifiers (see Using Special Modifiers). If `<file>` in the option contains a path (absolute or relative), the `OBJPATH` environment variable is ignored.

Example

```
-ObjN=a.out
```

The resulting object file is `a.out`. If the `OBJPATH` environment variable is set to `\src\obj`, the object file is `\src\obj\a.out. fibo.c -ObjN=%n.obj`

The resulting object file is `fibo.obj`.

```
myfile.c -ObjN=..\objects\_%n.obj
```

The object file is named relative to the current directory to `..\objects\_myfile.obj`. The OBJPATH environment variable is ignored because the `<file>` contains a path.

See also

OBJPATH: Object File Path

7.2.3.59 -Obsr: Generate Always Near Calls

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Obsr
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option forces the compiler to always generate near calls, i.e. use BSR instruction instead of a JSR in order to reduce code size. Without this option the compiler checks the range of the call to determine if a BSR can be generated instead of a JSR.

Example

```
extern int f(void);


int g(void) {


  return f();
```

The following listing shows an example without `-Obsr`:

Listing: Example - Without -obsr

```
  0000 b700               STA        __OVL_g_p0
  0002 45                 SHA

  0003 b700               STA        __OVL_g_14__PSID_75300004

  0005 42                 SLA

  0006 b701               STA        __OVL_g_14__PSID_75300004:1
```

```
   4:     return f();
 0008 a600            LDA         #__OVL_g_14__PSID_75300001
 000a bd0000          JSR         %FIX16(f)
 000d 4e000f          LDX         __OVL_g_p0
 0010 4e000e          MOV         __OVL_g_14__PSID_75300001,D[X]
 0013 2f              INCX
 0014 4e010e          MOV         __OVL_g_14__PSID_75300001:1,D[X]
 0017 b600            LDA         __OVL_g_14__PSID_75300004
 0019 45              SHA
 001a b601            LDA         __OVL_g_14__PSID_75300004:1
 001c 42              SLA
   5:   }
 001d be              RTS
```

The following listing shows an example with `-Obsr`:

Listing: Example - Without -obsr

```
 0000 b700            STA         __OVL_g_p0
 0002 45              SHA
 0003 b700            STA         __OVL_g_14__PSID_75300004
 0005 42              SLA
 0006 b701            STA         __OVL_g_14__PSID_75300004:1
   4:     return f();
 0008 a600            LDA         #__OVL_g_14__PSID_75300001
 000a ad00            BSR         PART_0_7(f)
 000c 4e000f          LDX         __OVL_g_p0
 000f 4e000e          MOV         __OVL_g_14__PSID_75300001,D[X]
 0012 2f              INCX
 0013 4e010e          MOV         __OVL_g_14__PSID_75300001:1,D[X]
 0016 b600            LDA         __OVL_g_14__PSID_75300004
 0018 45              SHA
 0019 b601            LDA         __OVL_g_14__PSID_75300004:1
 001b 42              SLA
   5:   }
 001c be              RTS
```

7.2.3.60 -Od: Disable Mid-level Optimizations

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Od [= <option Char> {<option Char>}]
```

Arguments

`<option Char>` is one of the following:

`a` : Disable mid level copy propagation

`b` : Disable mid level constant propagation

`c` : Disable mid level common subexpression elimination (CSE)

`d` : Disable mid level removing dead assignments

`e` : Disable mid level instruction combination

`f` : Disable mid level code motion

`g` : Disable mid level loop induction variable elimination

Default

None

Defines

None

Pragmas

None

Description

The backend of this compiler is based on the second generation intermediate code generator (SICG). All intermediate language and processor independent optimizations (cf. NULLSTONE) are performed by the SICG optimizer using the powerful static single assignment form (SSA form). The optimizations are switched off using `-od`. Currently four optimizations are implemented.

Examples

-Od disables all mid-level optimizations

-Od=d disables removing dead assignments only

-Od=cd disables removing dead assignments and CSE

See also

None

7.2.3.61 -Odb: Disable Mid-level Branch Optimizations

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
  -Odb [= <option Char> {<option Char>}]
```

Arguments

`<option Char>` is one of the following:

`a`: Disable mid level label rearranging

`b`: Disable mid level branch tail merging

`c`: Disable mid level loop hoisting

Default

None

Defines

None

Pragmas

None

Description

This option disables branch optimizations on the SSA form based on control flows. Label rearranging sorts all labels of the control flow to generate a minimum amount of branches.

Branch tail merging places common code into joining labels, as shown:

```
void fun(void) {void fun(void) {

if(cond) {if(cond) {

...

a = 0;} else {

} else {...

...}

a = 0;a = 0;

}}

}
```

Examples

-Odb disables all mid-level branch optimizations

-Odb=b disables only branch tail merging

See also

None

7.2.3.62 -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 solves this problem by allowing the Compiler to choose from a set of options to reach the smallest code size for every function. Without this feature, you must set some Compiler switches, which are fixed, over the whole compilation unit. With this feature, the Compiler is free to find the best option combination from a user-defined set for every function.

Standard merging rules applies also for this new option, e.g.,

```
-Or -OdocF="-Ocu|-Cu"
```

is the same as

```
-OrDOCF="-Ouc|-Cu"
```

The Compiler attempts to find the best option combination (of those specified) and evaluates all possible combinations of all specified sets, e.g., for the option shown in the following listing:

Listing: Example of dynamic option configuration

```
-W2 -OdocF="-Or|-Cni -Cu|-Oc"
```

The code sizes for following option combinations are evaluated:

```
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
```

Thus, if the more sets are specified, the longer the Compiler has to evaluate all combinations, e.g., for 5 sets 32 evaluations.

NOTE

Do not use this option to specify options with scope Application or Compilation Unit (such as memory model, float or double format, or object-file format) or options for the whole compilation unit (like inlining or macro definition). The generated functions may be incompatible for linking and executing.

Limitations:

- The maximum set of options set is limited to five, e.g., `-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 options compatible with functions. Future extensions will also provide this option for compilation units.

Example

```
-Odocf="-Or|-Cni"
```

7.2.3.63 -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, all calls of this function are replaced by the code of this function, if possible.

Using the `-Oi=c0` option 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 option extension [`=c<n>`] signifies that all functions with a size smaller than `<n>` are inlined. 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

7.2.3.64 -Ona: Disable alias checking

Group

OPTIMIZATIONS

Description

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.

7.2.3.65 -OiLib: Optimize Library Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-OiLib[=<arguments>]
```

Arguments

`<arguments>` are one or multiple of following suboptions:

`b`: inline calls to the `strlen()` function

`d`: inline calls to the `fabs()` or `fabsf()` functions

`e`: inline calls to the `memset()` function

`f`: inline calls to the `memcpy()` function

`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()` (e.g., `strlen("abc")`)
- `abs()` or `fabs()` (e.g., `` `f = fabs(f);' ``)
- `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 - 0xff
  - the ANSI library header file `<string.h>` is included

    An example for this is:

    ```
    (void)memset(&buf, 0, 50);
    ```

    In this case, the call to `memset()` is replaced with a call to `_memset_clear_8bitCount` present in the ANSI library (`string.c`).

- `memcpy()` is optimized only if:
  - the result is not used,
  - the size for the copy out is in the range `0` to `0xff`,
  - the ANSI library header file `<string.h>` is included.

    An example for this is:

    ```
    (void)memcpy(&buf, &buf2, 30);
    ```

    In this case the call to `memcpy()` is replaced with a call to `_memcpy_8bitCount` present in the ANSI library (`string.c`).

- `(char)1 << val` is replaced by `_PowOfTwo_8[val]` if `_PowOfTwo_8` is known at compile time. Similarly, for 16-bit and 32-bit shifts, the arrays `_PowOfTwo_16` and `_PowOfTwo_32` are

used. These constant arrays contain the values 1, 2, 4, 8, etc. They are declared in `hidef.h`. This optimization is performed only when optimizing for time.
- `-Oilib` without arguments: optimize calls to all supported library functions.

Example

Compiling the `f()` function with the `-Oilib=a` compiler 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

7.2.3.66 -OnB: Disable Branch Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
  -OnB
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option, all low-level branch optimizations are disabled.

Example

```
-OnB
```

See Also

None

7.2.3.67 -Onbsr: Disable far to near call optimization

Group

OPTIMIZATIONS

Description

Disables the JSR to BSR optimization. The compiler checks the range of the call to determine if a BSR can be generated instead of a JSR. If -Onbsr is used this optimization will be disabled.

7.2.3.68 -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

With usual startup code, all global variables are first set to 0 (zero out). If the definition contained an initialization value, this initialization value is copied to the variable (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 normally not used. Because the copy down needs much more space than the zero out, it usually contains copy down and zero out, zero out alone, or none of them.

In the ELF format, the object-file format permits optimization only if the whole array or structure is initialized with 0.

NOTE

This option controls the optimizations done in the compiler. However, the linker itself 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};
```

If this option is present, no copy down is generated for `i`.

For the `arr` array, it is not possible to separate initialization with 0 from initialization with 1.

7.2.3.69 -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 provides you with a way to switch OFF the replacement of CONST variable by the constant value.

Example

```
const int MyConst = 5;

int i;

void foo(void) {

  i = MyConst;

}
```

If the `-OnStVar` option is not set, the compiler replaces each occurrence of `MyConst` with its constant value 5; that is `i = MyConst` is transformed into `i = 5`;. The Memory or ROM needed for the MyConst constant variable is optimized as well. With the `-OnCstVar` option set, this optimization is avoided. This is logical only if you want unoptimized code.

7.2.3.70 -Onp: Disable Peephole Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Onp
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

If `-OnP` is specified, the whole peephole optimizer is disabled. The peephole optimizer removes useless loads and stores and applies pre and post increment addressing if possible.

Example

```
-Onp
```

SeeAlso

None

7.2.3.71 -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 problem is to generate the denser code while avoiding overhead whenever possible (this NULL check code is a good example). If you can ensure this pointer to a member will never be NULL, then this NULL check is useless. This option enables you to switch off the code generation for the NULL check.

Example

```
-OnPMNC
```

7.2.3.72 -Onr: Disable Reload from Register Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Onr
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option disables the low level register trace optimizations. If you use the option the code becomes more readable, but less optimal.

Example

```
-Onr
```

See Also

None

7.2.3.73 -Ont: Disable Tree Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ont[={%|&|*|+|-|/|0|1|7|8|9|?|^|a|b|c|d|e|

 f|g|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

`^`: Disable xor 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 bit or optimization

`~`: Disable bit neg optimization

Default

If `-Ont` is specified, all optimizations are disabled

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. This may be useful for debugging and for forcing the Compiler to produce `straightforward' code. Note that the optimizations below are just examples for the classes of optimizations.

If this option is set, the Compiler will not perform the following optimizations:

-Ont=~

Disable optimization of ` `~~i`' into ` `i`'

-Ont=|

Disable optimization of ` `i|0xffff`' into ` `0xffff`'

-Ont=w

Disable optimization of ` `if (1) i = 0;`' into ` `i = 0;`'

-Ont=v

Disable optimization of ` `do ... while(0)` into `...'

-Ont=u

Disable optimization of '`while (cond) break;`' into '`cond;`', provided there are no labels within the '*while*' statement list.

-Ont=t

Disable optimization of ` `for(;;) ...`' into ` `while(1) ...`'

-Ont=s

Disable optimization of ` `*&i`' into ` `i`'

-Ont=r

Disable optimization of ` `L<=4`' into 16-bit compares if 16-bit compares are better

-Ont=q

Reduction of long compares into int compares if int compares are better: (`-Ont=q` to disable it)

```
if (uL == 0)
```

is optimized into

```
if ((int)(uL>>16) == 0 && (int)uL == 0)
```

-Ont=p

Disable optimization of ` `(char)(long)i`' into ` `(char)i`'

-Ont=o

Disable optimization of ` `(short)(int)L`' into ` `(short)L`' if `short` and `int` have the same size

-Ont=n,-Ont=m:

Optimization of shift optimizations (<<, >>, `-Ont=n` or `-Ont=m` to disable it): Reduction of shift counts to unsigned char:

```
uL = uL1 >> uL2;
```

is optimized into:

```
uL = uL1 >> (unsigned char)uL2;
```

Optimization of zero shift counts:

```
uL = uL1 >> 0;
```

is optimized into:

```
uL = uL1;
```

Optimization of shift counts greater than the object to be shifted:

```
uL = uL1 >> 40;
```

is optimized into:

```
uL = 0L;
```

Strength reduction for operations followed by a cut operation:

```
ch = uL1 * uL2;
```

is optimized into:

```
ch = (char)uL1 * (char)uL2;
```

Replacing shift operations by load or store

```
i = uL >> 16;
```

is optimized into:

```
i = *(int *)(&uL);
```

Shift count reductions:

```
ch = uL >> 17;
```

is optimized into:

```
ch = (*(char *)(&uL)+1)>>1;
```

Optimization of shift combined with binary and:

```
ch = (uL >> 25) & 0x10;
```

is optimized into:

```
ch = ((*(char *)(&uL))>>1) & 0x10;
```

-Ont=l

Disable optimization removal of labels if not used

-Ont=i

Disable optimization of ` `&*p`' into ` `p`'

-Ont=j

This optimization transforms the syntax tree into an equivalent form in which more inlining cases can be done. This option only has an effect when inlining is enabled.

-Ont=h

Disable optimization of ` `-(-i)`' into ` `i`'

-Ont=f

Disable optimization of ` `(a==0)`' into ` `(!a)`'

-Ont=e

Disable optimization of ` `2*i`' into ` `i*2`'

-Ont=d

Disable optimization of ` `us` & `ui`' into ` `us` & `(unsigned short)ui`'

-Ont=c

Disable optimization of ` `if ((long)i)`' into ` `if (i)`'

-Ont=b

Disable optimization of ` `3+7`' into ` `10`'

-Ont=a

Disable optimization of last statement in function if result is not used

-Ont=^

Disable optimization of ` `i^0`' into ` `i`'

-Ont=?

Disable optimization of ` `i = (int)(cond ? L1:L2);`' into ` `i = cond ? (int)L1:(int)L2;`'

-Ont=9

Disable optimization of ` i=i;'

-Ont=8

Disable optimization of empty switch statement

-Ont=7

Disable optimization of ` (long)(char)L' into ` L'

-Ont=1

Disable optimization of ` a || 0' into ` a'

-Ont=0

Disable optimization of ` a && 1' into ` a'

-Ont=/

Disable optimization of ` a/1' into ` a'

-Ont=-

Disable optimization of ` a-0' into ` a'

-Ont=+

Disable optimization of ` a+0' into ` a'

-Ont=*

Disable optimization of ` a*1' into ` a'

-Ont=&

Disable optimization of ` a&0' into ` 0'

-Ont=%

Disable optimization of ` a%1' into ` 0'

Example

```
fibo.c -Ont
```

7.2.3.74 -Ontc: Disable tail call optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ontc
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

By default, the compiler replaces trailing calls (JSR/BSR) with JMP instructions if the function does not contain any other function calls. This allows the compiler to remove all the entry and exit code from the current function.

Example

```
void f1() {

  i++;

}

void f2() {

  f1();
}
```

The following listing shows an example without `-Ontc`:

Listing: Example - Without -Ontc

```
Function: f1
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst
    7:    i++;
  0000 3c01             INC        i:1
  0002 3602             BNE        L6
  0004 3c00             INC        i
  0006          L6:
    8:  }
  0006 be               RTS
    9:
   10:  void f2() {
Function: f2
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst
   11:    f1();
  0000 3000             BRA        PART_0_7(f1)
   12:  }
```

The following listing shows an example with `-Ontc`:

Listing: Example - With -Ontc

```
Function: f1
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst -Ontc
    7:    i++;
  0000 3c01             INC        i:1
  0002 3602             BNE        L6
  0004 3c00             INC        i
  0006          L6:
    8:  }
  0006 be                RTS
    9:
   10:  void f2() {
Function: f2
Source  : d:\junk\rs08\test.c
Options : -Lasm=%n.lst -Ontc
  0000 45                SHA
  0001 b700              STA        __OVL_f2_14__PSID_75300003
  0003 42                SLA
  0004 b701              STA        __OVL_f2_14__PSID_75300003:1
   11:    f1();
  0006 ad00              BSR        PART_0_7(f1)
  0008 b600              LDA        __OVL_f2_14__PSID_75300003
  000a 45                SHA
  000b b601              LDA        __OVL_f2_14__PSID_75300003:1
  000d 42                SLA
   12:  }
  000e be                RTS
   13:
```

7.2.3.75 -Ostk: Reuse Locals of Stack Frame

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Ostk
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option instructs the compiler to reuse the location of local variables/temporaries whenever possible. When used, the compiler analyzes which local variables are alive simultaneously. Based on that analysis the compiler chooses the best memory layout for for variables. Two or more variables may end up sharing the same memory location.

Example

```
TBD
```

7.2.3.76 -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

If escape sequences are used in macros, they are handled in an include directive similar to the way they are handled in a `printf()` instruction:

```
#define STRING "C:\myfile.h"
```

```
#include STRING
```

produces an error:

```
>> Illegal escape sequence
```

but used in:

```
printf(STRING);
```

```
produces a carriage return with line feed:
```

```
C:
```

```
myfile
```

If the `-Pe` option is used, escape sequences are ignored in strings that contain a DOS drive letter ('a - 'z', 'A' - 'Z') followed by a colon ' `:`' and a backslash ' `\`'.

When the `-Pe` option is enabled, the Compiler handles 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 `STRING` macro has to be the same for both 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" */

}
```

This option may be used to use macros for include files. This prevents escape sequence scanning in strings if the string starts with a DOS drive letter (`a` through `z` or `A` through `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
```

7.2.3.77 -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, it checks if this file to be included was already read. If so, the compiler ignores the `#include` directive. It is common practice to protect header files from multiple inclusion by conditional compilation, as shown in the following listing:

Listing: Conditional compilation

```
/* Header file myfile.h */

#ifndef _MY_FILE_H_

#define _MY_FILE_H_

/* ... content ... */

#endif /* _MY_FILE_H_ */
```

When the `#ifndef` and `#define` directives are issued, any header file content is read 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.

When all header files are protected in this manner, this option can safely accelerate the compilation.

This option must not be used when a header file must be included twice, e.g., the file contains macros which are set differently at the different inclusion times. In those instances, #pragma ONCE: Include Once is used to accelerate the inclusion of safe header files that do not contain macros of that nature.

Example

```
-Pio
```

7.2.3.78 -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, including the `default.env` file, the command line or whatever. When this option is given, the application opens the file as a configuration file. When `<file>` names only a directory instead of a file, the default name `project.ini` is appended. When the loading fails, a message box appears.

Example

```
compiler.exe -prod=project.ini
```

NOTE

Use the compiler executable name instead of `"compiler"`.

See also

Local Configuration File (usually project.ini)

7.2.3.79 -Qvtp: Qualifier for Virtual Table Pointers

Group

CODE GENERATION

Scope

Application

Syntax

```
  -Qvtp(none|far|near|paged)
```

Arguments

None

Default

```
  -Qvptnone
```

Defines

None

Pragmas

None

Description

Using a virtual function in C++ requires an additional pointer to virtual function tables. This pointer is not accessible and is generated by the compiler in every class object when virtual function tables are associated.

NOTE

Specifying an unsupported qualifier has no effect, e.g., using a `far` qualifier if the Backend or CPU does not support any `__far` data accesses.

Example

```
-QvtpFar
```

This sets the qualifier for virtual table pointers to `__far` enabling the virtual tables to be placed into a `__FAR_SEG` segment (if the Backend or CPU supports `__FAR_SEG` segments).

7.2.3.80 -Rp[t|e]: Large return Value Type

Group

OPTIMIZATIONS

Description

Compiler supports this option even though returning a 'large' return value may be not 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. Options are: Default, Large return value pointer, always with temporary (-Rpt), and Large return value pointer and temporary elimination (-Rpe).

7.2.3.81 -SIP2: Specify the address of the System Interrupt Pending 2 register

Specifies the address of the System Interrupt Pending 2 register. By default, it is set

to `0x1D`.

Group

CODE GENERATION

Scope

Compilation Unit

7.2.3.82 -T: Flexible Type Management

Group

LANGUAGE.

Scope

Application

Syntax

```
-T<Type Format>
```

Arguments

`<Type Format>`: See below

Default

Depends on target, see the Backend chapter

Defines

To deal with different type sizes, one of the following define groups in the following listing is predefined by the Compiler:

Listing: Predefined 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_DSP__

__DOUBLE_IS_IEEE32__

__DOUBLE_IS_DSP__

__LONG_DOUBLE_IS_IEEE32__

__LONG_DOUBLE_IS_DSP__

__LONG_LONG_DOUBLE_IS_IEEE32__

__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 option syntax is:

```
-T{<type><format>}
```

For `<type>`, one of the keys listed in the following table may be specified:

Table 7-9. 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, e.g., both `f` or `F` may be used for the type `float`.

The sign of the type `char` or of the enumeration type may be changed with a prefix placed before the key for the `char` key. Refer to the following table:

Table 7-10. Keys for Signed and Unsigned Prefixes

Sign Prefix	Key
`signed`	`s`
`unsigned`	`u`

The sign of the type `plain bitfield type` is changed with the options shown in the following table. Plain bitfields are bitfields defined or declared without an explicit signed or unsigned qualifier, e.g., `int field:3`. Using this option, you can specify if the `int` in the previous example is handled as `signed int` or as `unsigned int`. Note that this option may not be available on all targets. Also the default setting may vary. For more information, refer to the Sign of Plain Bitfields.

Table 7-11. Keys for Signed and Unsigned Bitfield Prefixes

Sign prefix	Key
plain signed bitfield	`bs`
plain unsigned bitfield	`bu`

For `<format>`, one of the keys in the following table can be specified.

Table 7-12. 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
DSP (32-bit)	0

Not all formats may be available for a target. See RS08 Backend for supported formats.

NOTE

At least one type for each basic size (1, 2, 4 bytes) has to be available. It is illegal if no type of any sort is not set to at least a size of one. See RS08 Backend for default settings.

NOTE

Enumeration types have the type `signed int` by default for ANSI-C compliance.

The `-Tpmo` option allows you to change the pointer to a member offset value type. The default setting is 16 bits. The pointer to the member offset is used for C++ pointer to members only.

Examples

```
-Tsc sets ´char´ to ´signed char´
```

and

```
-Tuc sets ´char´ to ´unsigned char´
```

Listing: -Tsc1s2i2L4LL4f2e2 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)

  enum with 16 bits (signed) (e2)
```

For integrity and compliance to ANSI, the following two rules must be true:

Listing: Restrictions

```
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

It is not permitted to set `char` to 16 bits and `int` to 8 bits.

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the Compiler are compiled with the standard type settings.

Also be careful if you change the type sizes for under or overflows, e.g., assigning a value too large to an object which is smaller now, as shown in the following example:

```
int i; /* -Ti1 int has been set to 8 bits! */


i = 0x1234; /* i will set to 0x34! */
```

Examples

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:

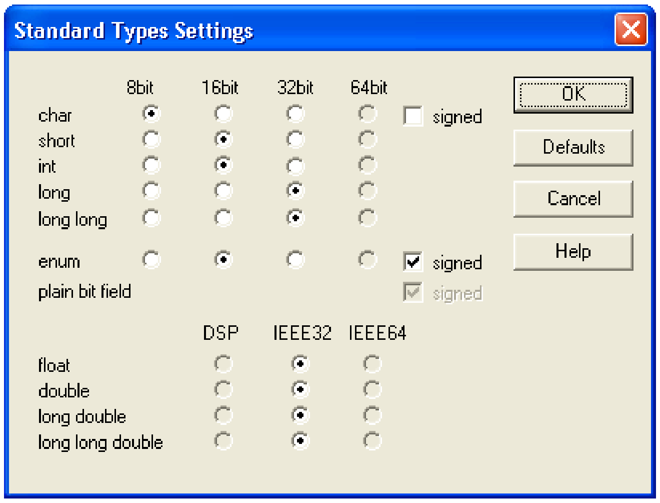
```
-Tuc1i4LL4
```

Setting float to IEEE32:

```
-Tf2
```

The `-Tvtd` option allows you to change the delta value type inside virtual function tables. The default setting is 16-bit.

Another way to set this option is using the dialog box in the Graphical User Interface:



	8bit	16bit	32bit	64bit	
char	●	○	○	○	☐ signed
short	○	●	○	○	
int	○	●	○	○	
long	○	○	●	○	
long long	○	○	●	○	
enum	○	●	○	○	☑ signed
plain bit field					☑ signed

	DSP	IEEE32	IEEE64
float	○	●	○
double	○	●	○
long double	○	●	○
long long double	○	●	○

Figure 7-3. Standard Types Settings Dialog Box

See also

Sign of Plain Bitfields

7.2.3.83 -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 component parts of the Compiler 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
```

7.2.3.84 -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

The application starts as a normal window if no arguments are given. If the application is started with arguments (e.g., from the maker to compile or link a file), then the application runs minimized to allow batch processing.

You can specify the behavior of the application using this option:

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

Example

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

7.2.3.85 -WErrFile: Create "err.log" Error File

Group

MESSAGES

Scope

Compilation Unit

Syntax

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

Arguments

None

Default

`err.log` is created or deleted

Defines

None

Pragmas

None

Description

The error feedback to the tools that are called is done with a return code. In 16-bit window environments, this was not possible. In the error case, an `err.log` file, with the numbers of errors written into it, was used to signal an error. To state no error, the `err.log` file was deleted. Using UNIX or WIN32, there is now a return code available. The `err.log` file is no longer needed when only UNIX or WIN32 applications are involved.

NOTE

The error file must be created in order to signal any errors if you use a 16-bit maker with this tool.

Example

```
-WErrFileOn
```

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

```
-WErrFileOff
```

The existing `err.log` file is not modified.

See also

-WStdout: Write to Standard Output

-WOutFile: Create Error Listing File

7.2.3.86 -Wmsg8x3: Cut File Names in Microsoft Format to 8.3

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
  -Wmsg8x3
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Some editors (e.g., early versions of WinEdit) expect the filename in the Microsoft message format (8.3 format). That means the filename can have, at most, eight characters with not more than a three-character extension. Longer filenames are possible when you use later versions of Windows. This option truncates the filename to the 8.3 format.

Example

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

With the `-Wmsg8x3` option set, the above message is:

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

See also

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

7.2.3.87 -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 also be specified in decimal.

Example

`-WmsgCE255` changes the error messages to blue

7.2.3.88 -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 also be specified in decimal.

Example

`-WmsgCF255` changes the fatal messages to blue

7.2.3.89 -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 also be specified in decimal.

Example

`-WmsgCI255` changes the information messages to blue

7.2.3.90 -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 also be specified in decimal.

Example

`-WmsgCU255` changes the user messages to blue

7.2.3.91 -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 also be specified in decimal.

Example

`-WmsgCW0` changes the warning messages to black

7.2.3.92 -WmsgFb (-WmsgFbi, -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

You can start the Compiler with additional arguments (e.g., files to be compiled together with Compiler options). If the Compiler has been started with arguments (e.g., from the Make Tool or with the appropriate argument from an external editor), the Compiler compiles the files in a batch mode. No Compiler window is visible and the Compiler terminates after job completion.

If the Compiler is in batch mode, the Compiler messages are written to a file instead of to the screen. This file contains only the compiler messages (see the examples in Listing: Message file formats (batch mode)).

The Compiler uses a Microsoft message format to write the Compiler messages (errors, warnings, information messages) if the compiler is in batch mode.

This option 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 because the compiler has to write more information into the message file.

Example

See the following listing for examples showing the differing message formats.

Listing: Message file formats (batch mode)

```
void foo(void) {
 int i, j;
 for (i=0;i<1;i++);
}
The Compiler may produce the following file if it is running in batch mode (e.g., started
from the Make tool):
X:\C.C(3): INFORMATION C2901: Unrolling loop
X:\C.C(2): INFORMATION C5702: j: declared in function foo but not referenced
Setting the format to verbose, more information is stored 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 foo(void) {
 int i, j;
 ^
INFORMATION C5702: j: declared in function foo but not referenced
```

See also

ERRORFILE: Error filename Specification environment variable

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

7.2.3.93 -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

The Compiler operates in the interactive mode (that is, a window is visible) if it is started without additional arguments (e.g., files to be compiled together with Compiler options).

The Compiler uses the verbose error file format to write the Compiler messages (errors, warnings, information messages).

This option changes the default format from the verbose format (with source, line and column information) to the Microsoft format (only line information).

NOTE

Using the Microsoft format may speed up the compilation because the compiler has to write less information to the screen.

Example

See the following listing for examples showing the differing message formats.

Listing: Message file formats (interactive mode)

```
void foo(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 (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

7.2.3.94 -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 formats listed in the following table are supported (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

Table 7-13. 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	`%`

Table continues on the next page...

Table 7-13. Message Format Specifiers (continued)

Format	Description	Example
`\n`	New line	
`%"`	A " if the filename, path, or extension contains a space	
`%'`	A ' if the filename, path, or 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 (-WmsgFbi, -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

7.2.3.95 -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 formats listed in the following table are supported (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:\sources\`
`%f`	Path and name	`X:\sources\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, path, or extension contains a space.	
`%'`	A ' if the filename, path, or extension contains a space	

Example

```
-WmsgFoi"%f%e(%l): %k %d: %m\n"
```

Produces a message in following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

ERRORFILE: Error filename Specification

-WmsgFb (-WmsgFbi, -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

7.2.3.96 -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 (e.g., if a message not related to a specific file). Then the message format string defined by `<string>` is used. The following table lists the supported formats.

Table 7-15. 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 following format:

```
information L10324: Linking successful
```

See also

ERRORFILE: Error filename Specification

Compiler options :

- -WmsgFb (-WmsgFbi, -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

7.2.3.97 -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 (e.g., if a message not related to a certain position). Then the message format string defined by `<string>` is used. 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 following format:

```
information L10324: Linking successful
```

Seealso

ERRORFILE: Error filename Specification

Compiler options:

- -WmsgFb (-WmsgFbi, -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

7.2.3.98 -WmsgNe: Maximum Number of Error Messages

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 number of error messages that are to be displayed 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

-WmsgNw: Maximum Number of Warning Messages

7.2.3.99 -WmsgNi: Maximum Number of Information Messages

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
  -WmsgNi<number>
```

Arguments

`<number>`: Maximum number of information messages

Default

`50`

Defines

None

Pragmas

None

Description

This option sets the amount of information messages that are logged.

Example

```
-WmsgNi10
```

Ten information messages logged

See also

Compiler options :

- -WmsgNe: Maximum Number of Error Messages
- -WmsgNw: Maximum Number of Warning Messages

7.2.3.100 -WmsgNu: Disable User Messages

Group

MESSAGES

Scope

None

Syntax

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

Arguments

`a`: Disable messages about include files

`b`: Disable messages about reading files

`c`: Disable messages about generated files

`d`: Disable messages about processing statistics

`e`: Disable informal messages

`t`: Display 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 messages that are not in the normal message category by reducing the amount of messages, and simplifying the error parsing of other tools.

a: Disables the application from generating information about all included files.

b: Disables messages about reading files (e.g., the files used as input) are disabled.

c: Disables messages informing about generated files.

d: Disables information about statistics (e.g., code size, RAM, or ROM usage).

e: Disables informal messages (e.g., memory model, floating point format).

t: Displays type of messages.

NOTE

Depending on the application, the Compiler may not recognize all suboptions. In this case they are ignored for compatibility.

Example

```
-WmsgNu=c
```

7.2.3.101 -WmsgNw: Maximum Number of Warning Messages

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

Compiler options:

- -WmsgNe: Maximum Number of Error Messages
- -WmsgNi: Maximum Number of Information Messages

7.2.3.102 -WmsgSd: Setting a Message to Disable

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSd<number>
```

Arguments

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

Default

None

Defines

None

Pragmas

None

Description

This option disables message 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

7.2.3.103 -WmsgSe: Setting a Message to Error

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSe<number>
```

Arguments

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

7.2.3.104 -WmsgSi: Setting a Message to Information

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSi<number>
```

Arguments

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

7.2.3.105 -WmsgSw: Setting a Message to Warning

Group

MESSAGES

Scope

Function

Syntax

```
-WmsgSw<number>
```

Arguments

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

7.2.3.106 -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 to create an error listing file. The error listing file contains a list of all messages and errors that are created during processing. It is possible to obtain this feedback without an explicit file because the text error feedback can now also be handled with pipes to the calling application. The name of the listing file is controlled by the environment variable ERRORFILE: Error filename Specification.

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

7.2.3.107 -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 issues an *ERROR* message instead of a *WARNING* message when an implicit declaration is encountered. This occurs if the Compiler does not have a prototype for the called function.

This option helps to prevent parameter-passing errors, which can only be detected at runtime. It requires that each function that is called is prototyped before use. The correct ANSI behavior is to assume that parameters are correct for the stated call.

This option is the same as using `-WmsgSe1801`.

Example

```
-Wpd

main() {

 char a, b;

 func(a, b); // <- Error here - only two parameters
}

func(a, b, c)

 char a, b, c;

{

 ...

}
```

See also

-WmsgSe: Setting a Message to Error

7.2.3.108 -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

The usual standard streams are available with Windows applications. Text written into them does not appear anywhere unless explicitly requested by the calling application. This option determines if error file text 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

7.2.3.109 -W1: Do Not Print Information Messages

Group

MESSAGES

Scope

Function

Syntax

```
-W1
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Inhibits printing INFORMATION messages. Only WARNINGs and ERROR messages are generated.

Example

```
-W1
```

See also

-WmsgNi: Maximum Number of Information Messages

7.2.3.110 -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 ERRORs are generated.

Example

```
-W2
```

See also

-WmsgNi: Maximum Number of Information Messages

-WmsgNw: Maximum Number of Warning Messages

Chapter 8
Compiler Predefined Macros

The ANSI standard for the C language requires the Compiler to predefine a couple of 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 has been given. Otherwise, additional keywords are accepted (not in the ANSI standard).

The following tables lists all Compiler defines with their associated names and options.

It is also possible to log all Compiler predefined defines to a file using the -Ldf: Log Predefined Defines to File compiler option.

8.1 Compiler Vendor Defines

The following table shows 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 how data is organized in memory: Little Endian (least significant byte first in memory) or Big Endian (most significant byte first in memory).

The Compiler provides the "endian" macros listed in the following table.

Table 8-4. Compiler macros for defining "endianness"

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 differences between little endian and big endian (refer to the following listing).

Listing: Little vs. big endian

```
unsigned long L  = 0x87654321;
unsigned short s = *(unsiged short*)&L; // BE: 0x8765,LE: 0x4321

unsigned char c  = *(unsinged char*)&L; // BE: 0x87,  LE: 0x21
```

8.4 Various Defines for Compiler Option Settings

The following table 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 preprocessor-valid 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 OPTIONs`.

As a parameter, only the option itself is tested and not a specific argument of an option.

For example:

```
#if __OPTION_ACTIVE__("-D") /* true if any -d option given
*/
```

```
#if __OPTION_ACTIVE__("-DABS") /* not allowed */
```

To check for a specific define use:

```
#if defined(ABS)
```

If the specified option cannot be checked to determine if it is active (i.e., options that no longer exist), the message "C1439: illegal pragma __OPTION_ACTIVE__" is issued.

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 the 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 defines 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`.
`__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 `unsigned char`.
`__WCHAR_T_IS_USHORT__`	Defined if the Compiler expects `wchar_t` in `stddef.h` to be `unsigned short`.
`__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 `unsigned long`.
`__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 `size_t` and `ptrdiff_t` standard types.

8.6.1 Macros for RS08

The following table shows the settings for the RS08 target:

Table 8-7. RS08 Compiler Defines

size_t Macro	Defined
`__SIZE_T_IS_UCHAR__`	always
`__SIZE_T_IS_USHORT__`	never
`__SIZE_T_IS_UINT__`	never
`__SIZE_T_IS_ULONG__`	never

Table 8-8. RS08 Compiler Pointer Difference Macros

ptrdiff_t Macro	Defined
`__PTRDIFF_T_IS_CHAR__`	always
`__PTRDIFF_T_IS_SHORT__`	never
`__PTRDIFF_T_IS_INT__`	never
`__PTRDIFF_T_IS_LONG__`	never

8.7 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. (for more information, refer to the backend chapter.) It is implementation-defined if the result is negative or positive when one of the operands is defined negative. This is illustrated in the following listing.

Listing: Effect of polarity upon 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 sections show how it is implemented in a backend.

8.7.1 Macros for RS08

Table 8-9. RS08 Compiler Modulo Operator Macros

Name	Defined
`__MODULO_IS_POSITIV__`	never

8.8 Object-File Format Defines

The Compiler defines some macros to identify the format (mainly used in the startup code if it is object file specific), depending on the specified object-file format option. The following table lists these defines.

Table 8-10. Object-file Format Defines

Name	Defined
`__ELF_OBJECT_FILE_FORMAT__`	`-F2`

8.9 Bitfield Defines

This section describes the defines and define groups available for the RS08 compiler.

8.9.1 Bitfield Allocation

The Compiler provides six predefined macros to distinguish between the different allocations:

Listing: Predefined 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 above-listed defines, you can write compatible code over different Compiler vendors even if the bitfield allocation differs. Note that the allocation order of bitfields is important (refer to the following listing).

Listing: Compatible bitfield allocation

```
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 allocation unit for bitfields in the Compiler is a byte, the allocation of memory for bitfields is always from the most significant BYTE to the least significant BYTE. For example, `__BITFIELD_MSBYTE_FIRST__` is defined as shown in the following listing:

Listing: __BITFIELD_MSBYTE_FIRST__ definition

```
/* example for __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 to is non-portable and, for the masking involved in unpacking individual fields, inefficient. It is recommended to use regular bit-and (&) and bit-or (|) operations for I/O port access.

8.9.2 Bitfield Type Reduction

The Compiler provides two predefined macros for enabled/disabled type size reduction. With type size reduction 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.

```
__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 to get optimized bitfields (refer to the following listing):

Listing: Compatible optimized bitfields

```
struct{
  long b1:4;

  long b2:4;

} myBitfield;

31                       7  3  0

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

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

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

7      3     0

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

|  b2  |  b1 |
-BfaTSRon

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

8.9.3 Sign of Plain Bitfields

For some architectures, the sign of a plain bitfield does not follow standard rules. Normally in the following (refer to the following listing):

Listing: Plain bitfield

```
struct _bits {
int myBits:3;
} bits;
```

`myBits` is signed, because plain `int` is also signed. To implement it as an unsigned bitfield, use the following code (refer to the following listing):

Listing: Unsigned bitfield

```
struct _bits {
unsigned int myBits:3;
} bits;
```

However, some architectures need to overwrite this behavior to be compliant to their EABI (Embedded Application Binary Interface). Under those circumstances, the -T: Flexible Type Management (if supported) is used. The option affects the following defines:

Listing: Affected Define Groups

```
define group__PLAIN_BITFIELD_IS_SIGNED__   /* defined if plain bitfield  is signed*/
__PLAIN_BITFIELD_IS_UNSIGNED__             /* defined if plain bitfield is unsigned */
```

8.9.3.1 Macros for RS08

The following table identifies the implementation in the Backend.

Table 8-11. RS08 Compiler-Backend Macros

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

8.9.4 Type Information Defines

The Flexible Type Management sets the defines to identify the type sizes. The following table lists these defines.

Table 8-12. Type Information Defines

Name	Defined
`__CHAR_IS_SIGNED__`	see `-T` option or Backend
`__CHAR_IS_UNSIGNED__`	see `-T` option or Backend
`__CHAR_IS_8BIT__`	see `-T` option or Backend
`__CHAR_IS_16BIT__`	see `-T` option or Backend

Table continues on the next page...

Table 8-12. Type Information Defines (continued)

Name	Defined
`__CHAR_IS_32BIT__`	see `-T` option or Backend
`__CHAR_IS_64BIT__`	see `-T` option or Backend
`__SHORT_IS_8BIT__`	see `-T` option or Backend
`__SHORT_IS_16BIT__`	see `-T` option or Backend
`__SHORT_IS_32BIT__`	see `-T` option or Backend
`__SHORT_IS_64BIT__`	see `-T` option or Backend
`__INT_IS_8BIT__`	see `-T` option or Backend
`__INT_IS_16BIT__`	see `-T` option or Backend
`__INT_IS_32BIT__`	see `-T` option or Backend
`__INT_IS_64BIT__`	see `-T` option or Backend
`__ENUM_IS_8BIT__`	see `-T` option or Backend
`__ENUM_IS_SIGNED__`	see `-T` option or Backend
`__ENUM_IS_UNSIGNED__`	see `-T` option or Backend
`__ENUM_IS_16BIT__`	see `-T` option or Backend
`__ENUM_IS_32BIT__`	see `-T` option or Backend
`__ENUM_IS_64BIT__`	see `-T` option or Backend
`__LONG_IS_8BIT__`	see `-T` option or Backend
`__LONG_IS_16BIT__`	see `-T` option or Backend
`__LONG_IS_32BIT__`	see `-T` option or Backend
`__LONG_IS_64BIT__`	see `-T` option or Backend
`__LONG_LONG_IS_8BIT__`	see `-T` option or Backend
`__LONG_LONG_IS_16BIT__`	see `-T` option or Backend
`__LONG_LONG_IS_32BIT__`	see `-T` option or Backend
`__LONG_LONG_IS_64BIT__`	see `-T` option or Backend
`__FLOAT_IS_IEEE32__`	see `-T` option or Backend
`__FLOAT_IS_DSP__`	see `-T` option or Backend
`__DOUBLE_IS_IEEE32__`	see `-T` option or Backend
`__DOUBLE_IS_DSP__`	see `-T` option or Backend
`__LONG_DOUBLE_IS_IEEE32__`	see `-T` option or Backend
`__LONG_DOUBLE_IS_DSP__`	see `-T` option or Backend
`__LONG_LONG_DOUBLE_IS_IEEE32__`	see `-T` option or Backend
`__LONG_LONG_DOUBLE_IS_DSP__`	see `-T` option or Backend
`__VTAB_DELTA_IS_8BIT__`	see `-T` option
`__VTAB_DELTA_IS_16BIT__`	see `-T` option
`__VTAB_DELTA_IS_32BIT__`	see `-T` option
`__VTAB_DELTA_IS_64BIT__`	see `-T` option
`__PLAIN_BITFIELD_IS_SIGNED__`	see option `-T` or Backend
`__PLAIN_BITFIELD_IS_UNSIGNED__`	see option `-T` or Backend

8.9.5 Freescale RS08-Specific Defines

The following table identifies implementations specific to the Backend.

Table 8-13. RS08 Back End Defines

Name	Defined
`__RS08__`	always
`__NO_RECURSION__`	always
`__PTR_SIZE_1__`	for small memory model
`__PTR_SIZE_2__`	for banked memory model
`__PTR_SIZE_3__`	never
`__PTR_SIZE_4__`	never

Chapter 9
Compiler Pragmas

A pragma, as shown in the below listing, defines how information is passed 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#pragma ONCE: Include Once and #pragma NO_STRING_CONSTR: No String Concatenation during preprocessing, which are valid for one file.

Listing: Pragma syntax

```
#pragma pragma_name [optional_arguments]
```

The value for `optional_arguments` 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` This is an invalid directive because two pragma names were combined into one pragma directive.

The following section describes all of the pragmas that affect the Frontend. All other pragmas affect only the code generation process and are described in the Backend section.

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	Scope of pragma where it is valid. (See below.)
Syntax	Specifies the syntax of the pragma in an EBNF format.
Synonym	Lists a synonym for the pragma or none, if a synonym does not exist.
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 how to use it.
Example	Gives an example of usage and effects of the pragma.
See also	Names related sections.

The following table is a description of the different scopes for 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 source file. For example, if the pragma is in a header file included from a source file, the pragma is not valid in the source file.
Compilation Unit	The pragma is valid from the current position until the end of the whole compilation unit. For example, if the pragma is in a header file included from a source file, it is valid in the source file too.
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 source file where such a header file is included 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.

9.1.1 #pragma CONST_SEG: Constant Data Segment Definition

Scope

Until the next `CONST_SEG` pragma

Syntax

```
#pragma CONST_SEG (<Modif>  <Name>|DEFAULT)
```

Synonym

CONST_SECTION

Arguments

Listing: Some strings which may be used for <Modif>

```
 __FAR_SEG    (compatibility alias: FAR) __PAGED_SEG
```

NOTE

Do not use a compatibility alias in new code. It only exists for backwards compatibility. Some of the compatibility alias names conflict with defines found in certain header files. Therefore, using them can cause hard to detect problems. Avoid using compatibility alias names.

The segment modifiers are backend-dependent. The __FAR_SEG and __PAGED_SEG modifier specifies a segment which is accessed with 16-bit addresses (RS08 addresses use up to 14 bits). The difference between the two is that objects defined in a segment marked by __PAGED_SEG are assumed not to cross page boundary. It is the user's responsibility to ensure this.

`<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. For more information, refer to the *Linker* section in *Build Tools Utilities* manual.

Default

```
DEFAULT
```

Description

This pragma allocates constant variables into a segment. The segment is then allocated in the link parameter file to specific addresses. The `CONST_SEG` pragma sets the current `const` segment. All constant data declarations are placed in this segment. The default segment is set with:

```
#pragma CONST_SEG DEFAULT
```

With the -Cc option set, constants are always allocated in constant segments in the ELF object-file format and after the first #pragma `CONST_SEG` (see -Cc: Allocate Const Objects into ROM).

The CONST_SEG pragma also affects constant data declarations as well as definitions. Ensure that all constant data declarations and definitions are in the same const segment.

Some compiler optimizations assume that objects having the same segment are placed together. Backends supporting banked data, 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: Put Next Variable Definition into ROM is active, the current const segment is not used.

The CONST_SECTION synonym has exactly the same meaning as CONST_SEG.

Examples

The following listing shows code that uses 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: same 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 different segments */
#pragma CONST_SEG __SHORT_SEG DEFAULT /* error: no modifiers for the
                                         DEFAULT segment are allowed
```

See also

RS08 Backend

Linker section of the Build Tools manual

#pragma DATA_SEG: Data Segment Definition

#pragma STRING_SEG: String Segment Definition

#pragma INTO_ROM: Put Next Variable Definition into ROM

-Cc: Allocate Const Objects into ROM

9.1.2 #pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing

Scope

Until the next `CREATE_ASM_LISTING` pragma

Syntax

```
#pragma CREATE_ASM_LISTING (ON|OFF)
```

Synonym

None

Arguments

`ON`: All following defines or objects are generated

`OFF`: All following defines or objects are not generated

Default

```
OFF
```

Description

This pragma determines if the following defines or objects are printed into the assembler include file.

A new file is generated only when the `-La` compiler option is specified together with a header file containing `#pragma CREATE_ASM_LISTING ON`.

Listing: 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

Generating Assembler Include Files (-La Compiler Option)

9.1.3 #pragma DATA_SEG: Data Segment Definition

Scope

Until the next `DATA_SEG` pragma

Syntax

```
#pragma DATA_SEG (<Modif>  <Name>|DEFAULT)
```

Synonym

```
DATA_SECTION
```

Arguments

Listing: Some of the strings which may be used for <Modif>

```
__TINY_SEG   __SHORT_SEG  (compatibility alias: SHORT)
  __DIRECT_SEG (compatibility alias: DIRECT)
__PAGED_SEG
  __FAR_SEG    (compatibility alias: FAR)
```

NOTE

Do not use a compatibility alias in new code. It only exists for backwards compatibility. Some of the compatibility alias names conflict with defines found in certain header files. Therefore, using them can cause problems which may be hard to detect. So avoid using compatibility alias names.

`__TINY_SEG`: specifies an operand that can be encoded on four bits (between 0x0 and 0xF). Use this modifier for frequently accessed global variables.

`__SHORT_SEG`: specifies an operand that can be encoded on five bits (between 0x0 and 0x1F). Use this modifier to access IO registers in the lower RS08 register bank.

`__DIRECT_SEG`: specifies an operand that can be encoded within the range 0x00 and 0xBF. Use this modifier to access global variables. If the operand does not fit into either four bits (__TINY_SEG) or five bits (__SHORT_SEG) then direct (8 bit) addressing is used.

__PAGED_SEG: specifies an operand that can be encoded within the range 0x100 and 0x3FF for read/write registers and within the range 0x00 and 0x3FFF for global read-only data. Use this modifier to access IO registers in the high RS08 register bank (read/write) or to access constant data. Objects allocated using __PAGED_SEG must not cross page boundaries.

__FAR_SEG: specifies an operand that can be encoded within the range 0x00 to 0x3FFF. Use this modifier to access large constant (read-only) data. Allocate FAR sections to multiple pages.

<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. For more information, refer to the linker manual.

Default

DEFAULT

Description

The DATA_SEG pragma allocates variables into the current data segment. This segment is used to place all variable declarations. Use this pragma to impose tiny or short addressing mode when accessing variables in the relevant sections. Set the default segment with:

```
#pragma DATA_SEG DEFAULT
```

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.

The RS08 compiler automatically allocates non-static local data into an OVERLAP section. The OVERLAP section uses the same address range as __DIRECT_SEG (0x00 - 0xBF).

Some instructions support tiny and short addressing. These instructions are encoded on one byte only rather than two bytes.

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 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 synonym DATA_SECTION means exactly same as DATA_SEG.

Example

The following listing shows source code that uses the DATA_SEG pragma.

Listing: Using the DATA_SEG pragma

```
/* in a header file */ #pragma DATA_SEG __TINY_SEG MyTinySection
char status;
#pragma DATA_SEG __ SHORT_SEG MyShortSection
unsigned char IOReg;
#pragma DATA_SEG DEFAULT
char temp;
#pragma DATA_SEG __ PAGED_SEG MyShortSection
unsigned char IOReg;
unsigned char *__paged io_ptr = &IOREG;
#pragma DATA_SEG __ PAGED_SEG MyPagedSection
const char table[10];
unsigned char *__paged tblptr = table;
#pragma DATA_SEG __ FAR_SEG MyFarSection
const char table[1000];
unsigned char *__far tblptr = table;
```

See also

RS08 Backend

Linker section of the Build Tools manual

#pragma CONST_SEG: Constant Data Segment Definition

#pragma STRING_SEG: String Segment Definition

#pragma INTO_ROM: Put Next Variable Definition into ROM

-Cc: Allocate Const Objects into ROM

9.1.4 #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 is the same as using the `-Oi` compiler option.

Listing: Using an INLINE pragma to inline a function

```
int i; #pragma INLINE
static void fun(void) {
  i = 12;
}
void main(void) {
  fun(); // results in inlining `i = 12;'
}
```

See also

#pragma NO_INLINE: Do not Inline next function definition

-Oi: Inlining

9.1.5 #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 subsequent segment pragma (`CONST_SEG`, `DATA_SEG`, `CODE_SEG`) disables the pragma.

NOTE

This pragma is only useful for the HIWARE object-file format (but not for ELF/DWARF).

NOTE

This pragma is to force a non-constant (meaning a normal `variable') object to be recognized as `` `const' `` by the compiler. 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.

Example

The following listing presents some examples which use 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

9.1.6 #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 name that is used and its content have no meaning other than each name can contain only one content string. However, multiple pragmas with different `NAME`s are legal.

For the Linker or for the Debugger, however, `NAME` might trigger some special functionality with `CONTENT` as an argument.

The Linker collects the `CONTENT` for every `NAME` from different object files and issues a message if `CONTENT` differs in 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 code shown in the following listing in a header file used by all compilation units, the Linker issues 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
```

9.1.7 #pragma LOOP_UNROLL: Force Loop Unrolling

Scope

Function

Syntax

```
#pragma LOOP_UNROLL
```

Synonym

None

Arguments

None

Default

None

Description

If this pragma is present, loop unrolling is performed for the next function. This is the same as setting the `-Cu` option for the following single function.

Listing: Using a LOOP_UNROLL pragma to unroll the 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

9.1.8 #pragma mark: Entry in CodeWarrior IDE Function List

Scope

Line

Syntax

```
#pragma mark {any text - no quote marks needed}
```

Synonym

None

Arguments

None

Default

None

Description

This pragma adds an entry into the function list of the CodeWarrior IDE. It also helps to introduce faster code lookups by providing a menu entry which directly jumps to a code position. With the special `#pragma mark -`, a separator line is inserted.

NOTE

The compiler does not actually handle this pragma. The compiler ignores this pragma. 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

For the 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++;

}
```

9.1.9 #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 messages to be set in the C1234 format

Default

None

Description

Messages are selectively set to an information message, a warning message, a disable message, or an error message.

NOTE

This pragma has no effect for messages which are produced during preprocessing. The reason is that the pragma parsing has to be done during normal source parsing but not during preprocessing.

NOTE

This pragma (as other pragmas) has to be specified outside of the function's scope. For example, it is not possible to change a message inside a function or for a part of a function.

Example

In the example shown in the following listing, parentheses ( ) were omitted.

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

Compiler options :

- -WmsgSd: Setting a Message to Disable
- -WmsgSe: Setting a Message to Error
- -WmsgSi: Setting a Message to Information
- -WmsgSw: Setting a Message to Warning

9.1.10 #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 entry code and is useful for inline assembler functions. The entry code prepares subsequent C code to run properly. It usually consists of pushing register arguments on the stack (if necessary), and allocating the stack space used for local variables and temporaries and storing callee saved registers according to the calling convention.

The main purpose of this pragma is for functions which contain only High-Level Inline (HLI) assembler code to suppress the compiler generated entry code.

One use of this pragma is in the startup function `_Startup`. At the start of this function the stack pointer is not yet defined. It has to be loaded by custom HLI code first.

NOTE

C code inside of a function compiled with `#pragma NO_ENTRY` generates independently of this pragma. The resulting C code may not work since it could access unallocated variables in memory.

This pragma is safe in functions with only HLI code. In functions that contain C code, using this pragma is a very advanced topic. Usually this pragma is used together with the pragma `NO_FRAME`.

Tip

Use a `#pragma NO_ENTRY` and a `#pragma NO_EXIT` with HLI-only functions to avoid generation of any additional frame instructions by the compiler.

The code generated in a function with `#pragma NO_ENTRY` may be unreliable. It is assumed that the user ensures correct memory use.

NOTE

Not all backends support this pragma. Some may still generate entry code even if this pragma is specified.

Example

The following listing shows how to use the `NO_ENTRY` pragma (along with others) to avoid any generated code by the compiler. 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

9.1.11 #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 two pragmas `NO_ENTRY` and `NO_EXIT` together avoid generation of any exit/ entry code. Functions written in High-Level Inline (HLI) assembler can therefore be used as custom entry and exit code.

The compiler can often deduce if a function does not return, but sometimes this is not possible. This pragma can then be used to avoid the generation of exit code.

Tip

Use a `#pragma NO_ENTRY` and a `#pragma NO_EXIT` with HLI-only functions to avoid generation of any additional frame instructions by the compiler.

The code generated in a function with `#pragma NO_EXIT` may not be safe. It is assumed that the user ensures correct memory usage.

NOTE

Not all backends support this pragma. Some may still generate exit code even if this pragma is specified.

Example

The following listing shows how to use the `NO_EXIT` pragma (along with others) to avoid any generated code by the compiler. 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

9.1.12 #pragma NO_FRAME: No Frame Code

Scope

Function

Syntax

```
#pragma NO_FRAME
```

Synonym

None

Arguments

None

Default

None

Description

This pragma is accepted for compatibility only. It is replaced by the `#pragma NO_ENTRY` and `#pragma NO_EXIT` pragmas.

For some compilers, using this pragma does not affect the generated code. Use the two pragmas `NO_ENTRY` and `NO_EXIT` instead (or in addition). When the compiler does consider this pragma, see the `#pragma NO_ENTRY` and `#pragma NO_EXIT` for restrictions that apply.

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` may be unreliable. It is assumed that the user ensures correct memory usage.

NOTE

Not all backends support this pragma. Some may still generate frame code even if this pragma is specified.

Example

The following listing shows how to use the `NO_FRAME` pragma (along with others) to avoid any generated code by the compiler. 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

9.1.13 #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. The pragma is used to avoid inlining a function which would be inlined because of the `-Oi` compiler option.

Listing: Use of #pragma NO_INLINE to prevent inlining a function.

```
// (With the -Oi option) int i;
#pragma NO_INLINE
static void foo(void) {
  i = 12;
}
void main(void) {
  foo(); // call is not inlined
}
```

See also

#pragma INLINE: Inline Next Function Definition

-Oi: Inlining

9.1.14 #pragma NO_LOOP_UNROLL: Disable Loop Unrolling

Scope

Function

Syntax

```
#pragma NO_LOOP_UNROLL
```

Synonym

None

Arguments

None

Default

None

Description

If this pragma is present, no loop unrolling is performed for the next function definition, even if 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

-Cu: Loop Unrolling

9.1.15 #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 a subroutine or return from an interrupt). This may be useful if you care about the return instruction itself or if the code has to fall through to the first instruction of the next function.

This pragma does not suppress the generation of the exit code at all (e.g., deallocation of local variables or compiler generated local variables). The pragma suppresses the generation of the return instruction.

NOTE

If this feature is used to fall through to the next function, smart linking has to be switched off in the Linker, because the next function may be not referenced from somewhere else. In addition, be careful 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 example in the following listing places some functions into a special named segment. All functions in this special code segment have to be called from an operating system every 2 seconds after each other. With the pragma some functions do not return. They fall directly to the next function to be called, 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

9.1.16 #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 is valid for the rest of the file in which it appears. It switches off the special handling of `#` as a string constructor. This is useful if 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, `#` is handled as a string constructor, which is not the desired behavior.

Listing: Using a NO_STRING_CONSTR pragma in order to alter the meaning of #

```
#pragma NO_STRING_CONSTR
#define HALT(x) __asm { \
 LOAD Reg,#3 \
 HALT x, #255\
 }
```

See also

Using the Immediate-Addressing Mode in HLI Assembler Macros

9.1.17 #pragma ONCE: Include Once

Scope

File

Syntax

```
#pragma ONCE
```

Synonym

None

Arguments

None

Default

None

Description

If this pragma appears in a header file, the file is opened and read only once. This increases compilation speed.

Example

```
#pragma ONCE
```

See also

-Pio: Include Files Only Once

9.1.18 #pragma OPTION: Additional Options

Scope

`Compilation Unit or until the nextOPTIONpragma`

Syntax

```
#pragma OPTION ADD [<Handle>] "<Option>"

#pragma OPTION DEL <Handle>

#pragma OPTION DEL ALL
```

Synonym

None

Arguments

`<Handle>`: An identifier - added options can selectively be deleted.

`<Option>`: A valid option string

Default

None

Description

Options are added inside of the source code while compiling a file.

The options given on the command line or in a configuration file cannot be changed in any way.

Additional options are added to the current ones with the `ADD` command. A handle may be given optionally.

The DEL command either removes all options with a specific handle. It also uses the ALL keyword to remove all added options regardless if they have a handle or not. Note that you only can remove options which were added previously with the OPTION ADD pragma.

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.
- These 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, and
  - -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 of declarations or definitions.

Example

The example in the following listing shows how to compile only 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 #pragma OPTION ADD */
/* are active again. */
```

The examples in the following listing show *improper* uses of the OPTION pragma.

Listing: Improper uses of the OPTION pragma

```
#pragma OPTION ADD -Or /* ERROR, quotes missing; use "-Or"  */ #pragma OPTION "-Or"   /*
ERROR, needs also the ADD keyword */
#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
```

9.1.19 #pragma REALLOC_OBJ: Object Reallocation

Scope

Compilation Unit

Syntax

```
  #pragma REALLOC_OBJ "segment" ["objfile"] object qualifier
```

Arguments

`segment`: Name of an already existing segment. This name must have been previously used by a segment pragma (`DATA_SEG`, `CODE_SEG`, `CONST_SEG`, or `STRING_SEG`).

`objfile`: Name of a object file. If specified, the object is assumed to have static linkage and to be defined in `objfile`. The name must be specified without alteration by the qualifier `__namemangle`.

`object`: Name of the object to be reallocated. Here the name as known to the Linker has to be specified.

`qualifier`: One of the following:

- `__near`,
- `__far`,
- `__paged`, or
- `__namemangle`.

Some of the qualifiers are only allowed to backends not supporting a specified qualifier generating this message. With the special `__namemangle` qualifier, the link name is changed so that the name of the reallocated object does not match the usual name. This feature detects when a `REALLOC_OBJ` pragma is not applied to all uses of one object.

Default

None

Description

This pragma reallocates an object (e.g., affecting its calling convention). This is used by the linker if the linker has to distribute objects over banks or segments in an automatic way (code distribution). The linker is able to generate an include file containing `#pragma REALLOC_OBJ` to tell the compiler how to change calling conventions for each object. See the Linker manual for details.

Example

The following listing uses the `REALLOC_OBJ` pragma to reallocate the `evaluate.o` object file.

Listing: Using the REALLOC_OBJ pragma to reallocate an object

```
#pragma REALLOC_OBJ "DISTRIBUTE1" ("evaluate.o") Eval_Plus __near
 __namemangle
```

See also

Message C420 in the Online Help

Linker section of the Build Tools manual

9.1.20 #pragma STRING_SEG: String Segment Definition

Scope

Until the next `STRING_SEG` pragma

Syntax

`#pragma STRING_SEG (<Modif><Name>|DEFAULT)`

Synonym

`STRING_SECTION`

Arguments

Listing: Some of the strings which may be used for <Modif>

```
 __FAR_SEG    (compatibility alias: FAR) __PAGED_SEG
```

NOTE

Do not use a compatibility alias in new code. It only exists for backwards compatibility. Some of the compatibility alias names conflict with defines found in certain header files. So avoid using compatibility alias names.

The `__SHORT_SEG` modifier specifies a segment that accesses using 8-bit addresses. The definitions of these segment modifiers are backend-dependent.

`<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. For information, refer to the linker manual.

Default

`DEFAULT`.

Description

This pragma allocates strings into a segment. Strings are allocated in the linker segment `STRINGS`. This pragma allocates strings in special segments. String segments also may have modifiers. This instructs the Compiler to access them in a special way when necessary.

Segments defined with the pragma `STRING_SEG` are treated by the linker like constant segments defined with #pragma `CONST_SEG`, so they are allocated in ROM areas.

The pragma `STRING_SEG` sets the current string segment. This segment is used to place all newly occurring strings.

NOTE

The linker may support a 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` has exactly the same meaning as `STRING_SEG`.

Example

The following listing is an example of 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 __FAR_SEG STRING_MEMORY char * __far p="String1";
```

```
void f(char * __far );
void main(void) {
 f("String2");
}
#pragma STRING_SEG DEFAULT
```

See also

RS08 Backend

Linker section of the Build Tools manual

#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 CompareOperator <Size> [<HashCode>]


CompareOperator: ==|!=|<|>|<=|>=
```

Arguments

`<Size>`: Size of the function to be used in a compare operation

`<HashCode>`: optional value specifying one specific code pattern.

Default

None

Description

This pragma checks the generated code. If the check fails, the message C3601 is issued.

The following parts are tested:

- Size of the function

  The compare operator and the size given as arguments are compared with the size of the function.

This feature checks that the compiler generates less code than a given boundary. Or, to be sure that certain code it can also be checked that the compiler produces more code than specified. To only check the hashcode, use a condition which is always TRUE, such as != 0.

- Hashcode

  The compiler produces a 16-bit hashcode from the produced 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 intentionally fail. Then copy the computed hashcode out of the body of the message C3601.

NOTE

The code generated by the compiler may change. If the test fails, it is often not certain that the topic chosen to be checked was wrong.

Examples

The following listing and Listing: Using a Test_Code pragma with the hashcode option 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 (refer to the following listing).

Listing: Using a Test_Code pragma with the hashcode option

```
/* 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 in the Online Help

9.1.22 #pragma TRAP_PROC: Mark function as interrupt function

Scope

Function Definition

Syntax

```
#pragma TRAP_PROC
```

Arguments

See Backend

Default

None

Description

This pragma marks a function to be an interrupt function. Because interrupt functions may need some special entry and exit code, this pragma has to be used for interrupt functions.

Do not use this pragma for declarations (e.g., in header files) because the pragma is valid for the next definition.

See the RS08 Backend chapter 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 chapter discusses features, restrictions, and further properties of the ANSI-C Compiler Frontend.

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. The reference document is *"American National Standard for Programming Languages - C"*, ANSI/ISO 9899-1990.

10.1.1 Keywords

See the following listing for the complete list of ANCSI-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 (refer to the following listing).

Listing: ANSI-C preprocessor directives

```
#if,
#ifdef,
#ifndef,
#else,
#elif,
#endif

     #define,
 #undef


     #include


     #pragma


     #error,
#line
```

The preprocessor operators `defined`, `#`, and `##` are also supported. There is a special non-ANSI directive `#warning` which is the same as `#error`, but issues only a warning message.

10.1.3 Language Extensions

There is a language extension in the Compiler for ANSI-C. You can use keywords to qualify pointers in order to distinguish them, or to mark interrupt routines.

The Compiler supports the following non-ANSI compliant keywords (see Backend if they are supported and for their semantics):

10.1.3.1 Pointer Qualifiers

Pointer qualifiers can be used to distinguish between different pointer types (e.g., for paging). Some of them are also used to specify the calling convention to be used (e.g., if banking is available).

Listing: Pointer qualifiers

```
__far  (alias
far)


__near (alias
near)

__paged
```

Far pointers contain a real RS08 address, that is, a linear address between 0 and 0x3FFF. A paged pointer is a 16-bit data entity containing a page number in the high byte and an address in the low byte. The address lies within the paging window, so no additional overhead is required to compute the value. Paged pointers cannot be used to access data across page boundaries. Use paged pointers whenever possible.

To allow portable programming between different CPUs (or if the target CPU does not support an additional keyword), you can include the defines listed below in the `hidef.h` header file.

Listing: far and near can be defined 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 listed keywords in the following listing do not conform to ANSI standards. However, they enable you 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)
```

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)

It is as well possible to use the 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 zeros or ones.

Listing: Demonstration of a binary constant

```
#define myBinaryConst 0b01011

  int i;

void main(void) {

  i = myBinaryConst;

}
```

10.1.3.4 Hexadecimal constants ($)

It is possible to use Hexadecimal constants inside HLI (High-Level Inline) Assembly. For example, instead of `0x1234` you can use `$1234`. Note that this is valid only for inline assembly.

10.1.3.5 #warning directive

The `#warning` directive, as shown in the following listing is used as it is similar to the `#error` directive.

Listing: #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)

You can assign global variables to specific addresses with the global variable address modifier. These variables are called absolute variables. They are useful for accessing memory mapped I/O ports and have the following syntax:

```
Declaration = <TypeSpec> <Declarator>
              [@<Address>|@"<Section>"] [= <Initializer>];
```

where:

- `<TypeSpec>` is the type specifier, e.g., `int`, `char`
- `<Declarator>` is the identifier of the global object, e.g., `i`, `glob`
- `<Address>` is the absolute address of the object, e.g., `0xff04`, `0x00+8`
- `<Initializer>` is the value to which the global variable is initialized.

A segment is created 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. Otherwise, there would be a linker error (overlapping segments). 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`. An alternate way to assign global variables to specific addresses is setting the `PLACEMENT` section in the Linker parameter file (see the following listing).

Listing: Assigning global variables to specific addresses

```
#pragma DATA_SEG [__SHORT_SEG] <segment_name>
```

An older method of accomplishing this is shown in the following listing.

Listing: Another means of assigning global variables to specific addresses

```
<segment_name> INTO  READ_ONLY <Address> ;
```

The following listing is an example using correctly and incorrectly the global variable address modifier and the following listing is a possible PRM file that corresponds with the example Listing.

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 Linker 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"

Sometimes it is useful to have the variable directly allocated in a named segment instead of using a `#pragma`. The following listing is an example of how to do this.

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";
```

So with some pragmas in a common header file and with another definition for the macro, it is possible to allocate variables depending on a 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, e.g., `int` or `char`
- `<Declarator>` is the identifier of your global object, e.g., `i`, `glob`
- `<Section>` is the section name. Define it in the link parameter file as well (e.g., `MyDataSection`).
- `<Initializer>` is the value to which the global variable is initialized.

The section name used has to be known at the declaration time by a previous section pragma.

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 good practice
#pragma CONST_SEC  DEFAULT       // not necessary,
                                 // but 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
```

10.1.3.7.1 Absolute Functions

Sometimes it is useful to call an absolute function (e.g., a special function in ROM). The following listing is a simple example of calling an absolute function using normal ANSI-C.

Listing: Absolute function

```
#define erase ((void(*)(void))(0xfc06))

void main(void) {

  erase(); /* call function at address 0xfc06 */

}
```

10.1.3.7.2 Absolute Variables and Linking

Special attention is needed if absolute variables are involved in the linker's link process.

If an absolute object is not referenced by the application, it is always linked using the ELF/DWARF format. To force linking, switch off smart linking in the Linker, or using the `ENTRIES` command in the linker parameter file.

NOTE

Interrupt vector entries are always linked.

The example in the following listing shows how the linker handles 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;/* ELF: download! */

static       char L@0x13;      /* no zero out! */

static       char M@0x14 = 3;  /* copy down    */

static const char N@0x15 = 4;  /* ELF: download */

void interrupt 2 MyISRfct(void) {} /* download, always linked! */

  /* vector number two is downloaded with &MyISRfct */

void foo(char *p) {} /* download */

void main(void) { /* download */

  foo(&i); foo(&j); foo(&k);

  foo(&I); foo(&J); foo(&K);

  foo(&L); foo(&M); foo(&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.

10.1.3.8 The __far Keyword

The keyword `far` is a synonym for `__far`, which is not allowed when the -Ansi: Strict ANSI compiler option is present.

A __far pointer allows access to the whole memory range supported by the processor, not just to the default data page. You can use it to access memory mapped I/O registers that are not 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.

10.1.3.8.1 Using the __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;` not `char __far *p;` The second choice will not work.

10.1.3.8.2 __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);
```

There are two possible uses 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);
```

10.1.3.8.3 __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`".

10.1.3.8.4 __far and Global Variables

The `__far` keyword can also be used 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 perform the same addressing mode for this 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 if you mix `__far` declarations or definitions within a non-`__FAR_SEG` data segment. Assuming that `__FAR_SEG` segments have extended addressing mode and normal segments have direct addressing mode, the following listing and the following listing clarify this behavior:

Listing: OK - 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 MyDirectSet

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.

10.1.3.8.5 __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:C1145: Class '<class_name>', qualified with modifier__far, inherits from non-qualified base class'<class_name>'

10.1.3.8.6 __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;

};
```

10.1.3.8.7 Using the __far Keyword for Functions

A special calling convention is specified for the `__far` keyword. The `__far` keyword is specified in front of the function identifier:

```
  void __far f(void);
```

If the function returns a pointer, the `__far` keyword must be written in front of 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 if the calling convention was not specified. 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.

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
`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`

10.1.3.9 __near Keyword

The near keyword is a synonym for `__near`. The `near` keyword is only allowed when the -Ansi: Strict ANSI compiler option is present.

The `__near` keyword can be used instead of the `__far` keyword. Use it in situations where non-qualified pointers are `__far` and you want to specify an explicit `__near` access or when you must explicitly specify the `__near` calling convention.

The `__near` keyword uses two semantic variations. Either it specifies a small size of a function or data pointers 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`
`void __far *__near (*__near f)(void)`	OK	`__near` pointer to `__far` function returning a `__far` pointer to `void`

10.1.3.10 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 the other case, the `__far` pointer is clipped to a normal pointer (i.e., the page part is discarded).

10.1.3.11 __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`, e.g., 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]))
```

10.1.3.12 __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.

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 `va_arg` macro is written the way that `f()` 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.

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

It is not possible in ANSI-C to distinguish a 1-byte structure without alignment or padding from a character variable in a `va_arg` macro. These need a different space on the open parameter calls stack for some processors.

10.1.3.13 interrupt keyword

The `__interrupt` keyword is a synonym for interrupt, which is allowed when the -Ansi: Strict ANSI compiler option is present.

NOTE

Not all Backends support this keyword. See the Non-ANSI Keywords section in RS08 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 nonstandard interrupt keyword.

Use the nonstandard interrupt keyword like any other type qualifier. It specifies a function to be 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 it is not followed by any number, the interrupt keyword has the same effect as the `TRAP_PROC` pragma. It specifies a function to be an interrupt routine. However, the number of the interrupt vector must be associated 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
```

10.1.3.14 __asm Keyword

The Compiler supports target processor instructions inside of C functions.

The `asm` keyword is a synonym for `__asm`, which is allowed when the -Ansi: Strict ANSI compiler option is not present.

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
```

10.1.4 Implementation-Defined Behavior

The ANSI standard contains a couple of places where the behavior of a particular Compiler is left undefined. It is possible for different Compilers to implement certain features in different ways, even if they all comply with the ANSI-C standard. Subsequently, the following discuss those points and the behavior implemented by the Compiler.

10.1.4.1 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, an arithmetic right shift is performed.

10.1.4.2 Initialization of Aggregates with non Constants

The initialization of aggregates 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 */
}
```

10.1.4.3 Sign of char

The ANSI-C standard leaves it open, whether the data type `char` is signed or unsigned.

10.1.4.4 Division and Modulus

The results of the `"/"` and `"%"` operators are also not properly defined for signed arithmetic operations unless both operands are positive.

NOTE

The way a Compiler implements `"/"` and `"%"` for negative operands is determined by the hardware implementation of the target's division instructions.

10.1.5 Translation Limitations

This section describes the internal limitations of the Compiler. Some 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, or if it is a 16-bit Compiler running on a 16-bit platform (e.g., Windows for Workgroups).

The ANSI-C column in the table below describes the recommended limitations of ANSI-C (5.2.4.1 in ISO/IEC 9899:1990 (E)) standard. These quantities are only guidelines and do not determine compliance. The `Implementation' column shows the actual implementation value and the possible message number. `-' means that there is no information available for this topic and `n/a' denotes that this topic is not available.

Table 10-3. Translation Limitations (ANSI)

Limitation	Implementation	ANSI-C
Nesting levels of compound statements, iteration control structures, and selection control structures	256 (C1808)	15
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

Table continues on the next page...

Table 10-3. Translation Limitations (ANSI) (continued)

Limitation	Implementation	ANSI-C
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
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 10-4. 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	Expansion of recursive macros 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.

Table continues on the next page...

Table 10-4. Translation Limitations (non-ANSI) (continued)

Limitation	Description
Numbers	Maximum of 655,360,000 different numbers for a single compilation unit (C2700, C3302).
Goto	M68k 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 (e.g., by using CSE).
Code Size	Code size is limited to 32KB for each single function.
filenames	Maximum length for filenames (including path) are 128 characters for 16-bit applications or 256 for Win32 applications. UNIX versions support filenames without path of 64 characters in length and 256 in the path. Paths may be 96 characters on 16-bit PC versions, 192 on UNIX versions or 256 on 32-bit PC versions.

10.2 ANSI-C Standard

This section provides a short overview about the implementation (see also ANSI Standard 6.2) of the ANSI-C conversion rules.

10.2.1 Integral Promotions

You may use a `char`, a `short int`, or an `int` bitfield, or their signed or unsigned varieties, or an enum type, in an expression wherever an int or `unsigned int` 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 `unsigned int`. Integral promotions preserve value including sign.

10.2.2 Signed and Unsigned Integers

Promoting a signed integer type to another signed integer type of greater size requires `"sign extension"`: 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.

Then 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 `long int` and the other has type `unsigned int`, if a `long int` can represent all values of an `unsigned int`, the operand of type `unsigned int` is converted to `long int`; if a `long int` cannot represent all the values of an `unsigned int`, both operands are converted to `unsigned long int`.
- If either operand has type `long int`, the other operand is converted to `long int`.
- If either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
- Both operands have type `int`.

10.2.4 Order of Operand Evaluation

The priority order of operators and their associativity is listed in the following listing.

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.

10.2.4.1 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)
```

Furthermore, is the same as:

if (0) => Therefore, the if condition is always `false'.

Hint: use brackets if you are not sure about associativity!

10.2.5 Rules for Standard-Type Sizes

In ANSI-C, enumerations have the type of `int`. In this implementation they have to be smaller than or equal to `int`.

Listing: Size relationships among the 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-Type Formats

The RS08 compiler supports only the IEEE32 floating point format. The figure below shows this format.

Floats are implemented as IEEE32. This may vary for a specific Backend, or possibly, both formats may not be supported.

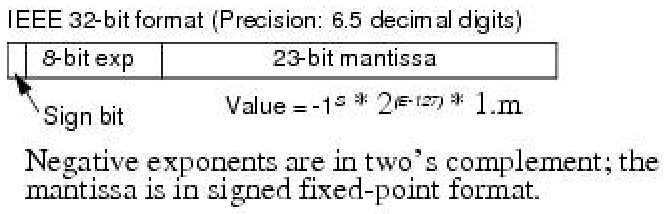

IEEE 32-bit format (Precision: 6.5 decimal digits)
8-bit exp
23-bit mantissa
Sign bit
Value = -1^s * 2^(E-127) * 1.m
Negative exponents are in two's complement; the mantissa is in signed fixed-point format.

Figure 10-1. Floating-point formats

10.3.1 Floating-Point Representation of 500.0 for IEEE

First, convert `500.0` from the decimal representation to a representation with base 2:

```
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. So 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 always have positive values:

```
exp (500.0,IEEE32) = 8+127 == 87 (hex) ==  10000111 (bin)
```

The fields must be put together as shown the following listing:

Listing: Representation of decimal 500.0 in IEEE32

```
500.0 (dec) =
  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 IEEE32 representation of decimal -500 is shown in the following listing.

Listing: Representation of decimal -500.0 in IEEE32

```
-500.0 (dec) =

  1 (sign) 10000111 (exponent)

  11111010000000000000000 (mantissa) (IEEE32 as bin)

  1100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)

  C3 fa 00 00 (IEEE32 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 do exist. For more information, refer to the *IEEE standard* documentation. Except for the 0 (zero) and -0 (minus zero) special formats, not all special formats may be supported for specific backends.

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 absolute global objects are not eliminated. See the following listing for one reason to use a volatile declaration.

Listing: Using volatile 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 (&) and bit-or (|) operations for I/O port access.

The maximum width of bitfields is Backend-dependent (see RS08 Backend for details), in that plain `int` bitfields are signed. A bitfield never crosses a word (2 bytes) boundary. As stated in Kernighan and Ritchie's *The C Programming Language*, 2nd 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.

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 of -1 or 0. Consider the following example shown in the following listing).

Listing: Testing a signed bitfield as being 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)` is always false. The test `(b.b0 == -1)` is performed as expected. This substitution is not ANSI compatible and will not be performed when the -Ansi: Strict ANSI compiler option is active.

The correct way to specify this is with an unsigned bitfield. Unsigned bitfields have the values 0 or 1.

Listing: Using unsigned bitfields

```
typedef struct _B {   unsigned b0: 1;
} B;
  B b;
  if (b.b0 == 1) ...
```

Because `b0` is an unsigned bitfield having the values 0 or 1, the test `(b.b0 == 1)` is correct.

10.5.1.1 Recommendations

In order to save memory, it is recommended to implement globally accessible boolean flags as unsigned bitfields of width 1. However, using bitfields for other purposes is not recommended 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.

10.6 Segmentation

The Linker supports the concept of segments in that the memory space may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions which then are allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

Listing: Syntax for 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|CompatSegmentMod

CompatSegmentMod: DIRECT|NEAR|CODE|FAR|BIT|

                  Y_BASED|Z_BASED|DPAGE|PPAGE|

                  EPAGE|RPAGE|GPAGE|PIC
```

Because there are two basic types of segments, code and data segments, there are also two pragmas to specify segments:

```
#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 are valid until the next pragma of the same kind is encountered.

In the ELF object file format, constants are always put into a constant segment.

Strings are put into the segment STRINGS until a pragma STRING_SEG is specified. After this pragma, all strings are allocated into this constant segment. The linker then treats this segment like any other constant segment.

If no segment is specified, 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
```

Segments may also be declared as `__SHORT_SEG` by inserting the keyword `__SHORT_SEG` just before the segment name (with the exception of the predefined segment `DEFAULT` - this segment cannot be qualified with `__SHORT_SEG`). This makes the Compiler use short (i.e., 8 bits or 16 bits, depending on the Backend) absolute addresses 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

The default code and data segments may not be declared as __SHORT_SEG.

The meaning of the other segment modifiers, such as __NEAR_SEG and __FAR_SEG, are backend-specific. Modifiers that are not supported by the backend are ignored.

The segment pragmas also have an effect on static local variables. Static local variables are local variables with the `static' flag set. They are in fact normal global variables but with scope only to the function in which they are defined:

```
#pragma DATA_SEG MySeg

static char foo(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), all constants are placed into the section .rodata by default unless #pragma CONST_SEG is used.

NOTE

There are aliases to satisfy the ELF naming convention 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 called optimizations. This section provides a short overview about the most important optimizations.

10.7.1 Peephole Optimizer

A peephole optimizer is a simple optimizer in a Compiler. A peephole optimizer tries to optimize specific code patterns on speed or code size. After recognizing these specific patterns, they are replaced by other optimized patterns.

After code is generated by the backend of an optimizing Compiler, it is still possible that code patterns may result that are still capable of being optimized. The optimizations of the peephole optimizer are highly backend-dependent because the peephole optimizer was implemented with characteristic code patterns of the backend in mind.

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. There are optimizations (e.g., removing of a branch to the next instructions) that are removed by the peephole optimizer, though they could have been removed by the branch optimizer as well. Such simple branch optimizations are performed in the peephole optimizer to reach new optimizable states.

10.7.2 Strength Reduction

Strength reduction is an optimization that strives to replace expensive operations by cheaper ones, where the cost factor is either execution time or code size. Examples are the replacement of multiplication and division by constant powers of two with left or right shifts.

NOTE

The compiler can only replace a division by two using a shift operation if either the target division is implemented the way that -1/2 == -1, or if the value to be divided is unsigned. The result is different for negative values. To give the compiler the possibility to use a shift, ensure that the C source code already contains a shift, or that the value to be shifted is unsigned.

10.7.3 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.

10.7.4 Branch Optimizations

This optimization tries to minimize the span of branch instructions. The Compiler will never generate a long branch where a short branch would have sufficed. Also, branches to branches may be resolved into two branches to the same target. Redundant branches (e.g., a branch to the instruction immediately following it) may be removed.

10.7.5 Dead-Code Elimination

The Compiler removes dead assignments while generating code. In some programs it may find additional cases of expressions that are not used.

10.7.6 Constant-Variable Optimization

If a constant non-volatile variable is used in any expression, the Compiler replaces it by the constant value it holds. This needs less code than taking the object itself.

The constant non-volatile object itself is removed if there is no expression taking the address of it (take note of `ci` in the following listing). This results in using 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
}
```

Global constant non-volatile variables are not removed. Their use in expressions are replaced by the constant value they hold.

Constant non-volatile arrays are also optimized (take note of `array[]` in the following listing).

Listing: Example demonstrating the optimization of a constant, non-volatile array

```
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.

10.7.7.1 Switch Statements

Efficient translation of switch statements is mandatory for any C Compiler. The Compiler applies different strategies, i.e., 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 10-5. 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 the equivalent to unrolling a binary search loop of a sorted jump table and 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	In such cases, the Compiler creates a table plus a call of a switch processor. There are two different switch processors. If there are a lot of labels with more or less consecutive values, a direct jump table is used. If the label values are scattered, a binary search table is used.
Mixed Strategy	Finally, there may be switches having clusters of label values separated by other labels with scattered values. In this case, a mixed strategy is applied, generating branch trees or search tables for the scattered labels and direct jump tables for the clusters.

10.7.7.2 Absolute Values

Another example for optimization on a higher level is the calculation of absolute values. In C, the programmer has to write something on the order of:

```
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 has to be cleared.

10.7.7.3 Combined Assignments

The Compiler can also recognize 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` because `const` and `volatile` are very common for Embedded Programming.

Consider the following example:

```
int i;
const int ci;
```

The above definitions are: a `normal' variable `i` and a constant variable `ci`. Each are placed into ROM. Note that for C++, the constant `ci` must be initialized.

```
int *ip;
const int *cip;
```

`ip` is a pointer to an `int`, where `cip` is a pointer to a `const int`.

```
int *const icp;
 const int *const cicp;
```

`icp` is a `const` pointer to an `int`, where `cicp` is a `const` pointer to a `const int`.

It helps if you know that the qualifier for such pointers is always on the right side of the `*`. Another way is to read the source from right to left.

You can express this rule in the same way to volatile. Consider the following example of an `array of five constant pointers to volatile integers':

```
volatile int *const arr[5];
```

`arr` is an array of five constant pointers pointing to volatile integers. Because the array itself is constant, it is put into ROM. It does not matter if the array is constant or not regarding where the pointers point to. Consider the next example:

```
const char *const *buf[] = {&a, &b};
```

Because the array of pointers is initialized, the array is not constant. `buf' is a (non-constant) array of two pointers to constant pointers which points to constant characters. Thus `buf' cannot be placed into ROM by the Compiler or Linker.

Consider a constant array of five ordinary function pointers. Assuming that:

```
void (*fp)(void);
```

is a function pointer `fp' returning void and having void as parameter, you can define it with:

```
void (*fparr[5])(void);
```

It is also possible to 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 an additional one:

```
void (*const fp[3])(void);
```

This is an array of three constant function pointers, having void as parameter and returning void. `fp' is allocated 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;
   }
```

It is actually a function called `func`. This `func` has one function pointer argument called `f`. The return value is more complicated in this example. It is actually a function pointer of a complex type. Here we do not explain where to put a `const` so that the destination of the returned pointer cannot be modified. Alternately, the same function is written more simply using typedefs:

```
typedef int (*funcType1) (void);



typedef int (* funcType2) (funcType1);



typedef funcType2 (* funcType3) (funcType1);
```

```
funcType3* func0(funcType1 f) {

  return 0;

}
```

Now, the places of the const becomes obvious. Just behind the `*` in `funcType3`:

```
typedef funcType2 (* const constFuncType3) (funcType1);

constFuncType3* func1(funcType1 f) {

  return 0;

}
```

By the way, also in the first version here is the place where to put the `const`:

```
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 you are working with the HLI assembler. Because the HLI assembler is heavily Backend-dependent, the following example uses a pseudo Assembler Language:

Listing: Example - Pseudo Assembler Language

```
  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 are only used as a possible example. For real applications, you must replace the above pseudo HLI instructions with the HLI instructions for your target.

10.9.1 Defining a Macro

An HLI assembler macro is defined by using the `define` preprocessor directive.

For example, a macro could be defined to clear 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.

Listing: Invoking the ClearR0 macro.

```
  ClearR0;
```

And then the preprocessor expands the macro.

Listing: Preprocessor expansion of ClearR0.

```
  { __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.

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 macro is invoked in the following way in the source code:

```
  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 macro is invoked in the following way in the source code.

Listing: Calling the ClearR0andR1 macro

```
ClearR0andR1;
```

The preprocessor expands the macro.

Listing: Preprocessor expansion of 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 that uses the var parameter.

Listing: Clear1 macro definition.

```
/* This macro initializes the specified variable to 0.*/
#define Clear1(var) {__asm CLR var;}
```

Listing: Invoking the Clear1 macro in the source code

```
Clear1(var1);
```

Listing: The preprocessor expands the Clear1 macro

```
{__asm CLR var1 ; };
```

10.9.3 Using the Immediate-Addressing Mode in HLI Assembler Macros

There may be one ambiguity if you are using the immediate addressing mode inside of a macro.

For the ANSI-C preprocessor, the symbol # inside of a macro has a specific meaning (string constructor).

Using #pragma NO_STRING_CONSTR: No String Concatenation during preprocessing, instructs the Compiler that in all the macros defined afterward, the instructions remain unchanged wherever 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;}
```

Listing: Invoking the Clear2 macro in the source code

```
Clear2(var1);
```

Listing: The preprocessor expands the Clear2 macro

```
{ __asm LOAD #0,Reg0;__asm STORE Reg0,var1; };
```

10.9.4 Generating Unique Labels in HLI Assembler Macros

When some labels are defined in HLI Assembler Macros, if you invoke the same macro twice in the same function, the ANSI C preprocessor generates the same label twice (once in each macro expansion). Use the special string concatenation operator of the ANSI-C preprocessor (` ##') in order to generate unique labels. See 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; }
```

Listing: 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, an `lp2` label is created.

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 direct accessing of the hardware.

The compiler cannot read the include files of the assembler, and the assembler cannot read the header files of the compiler.

The assembler's include file output of the compiler lets both tools use one single source to share constants, variables or labels, and even structure fields.

The compiler writes an output file in the format of the assembler which contains all information needed of a C header file.

The current implementation supports the following mappings:

- Macros

  C defines are translated to assembler `EQU` directives.

- enum values

  C `enum` values are translated to `EQU` directives.

- C types

  The size of any type and the offset of structure fields is generated for all `typedefs`. For bitfield structure fields, the bit offset and the bit size are also generated.

- Functions

  For each function an `XREF` entry is generated.

- Variables

  C Variables are generated with an `XREF`. In addition, for structures or unions all fields are defined with an `EQU` directive.

- Comments

  C style comments (`/* ... */`) are included as assembler comments (`;...`).

10.9.5.1 General

A header file must be specially prepared to generate the assembler include file.

Listing: A pragma anywhere in the header file can enable assembler output

```
#pragma CREATE_ASM_LISTING ON
```

Only macro definitions and declarations behind this pragma are generated. The compiler stops generating future 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. Care must be taken for macros. 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`.

10.9.5.1.1 Example

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.

Usually the assembler include file is not created every time the compiler reads the header file. It is only created in a separate pass when the header file has changed significantly. The `-La` option is only specified when the compiler must generate `a.inc`. If `-La` is always present, `a.inc` is always generated. A make tool will always restart the assembler because the assembler files depend on `a.inc`. Such a makefile might be similar to:

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 of 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 matters for the assembler. If the assembler has problems with the order of `EQU` directives in a generated file, the corresponding header file must be changed accordingly.

10.9.5.2 Macros

The translation of defines is done lexically and not semantically, so 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: Assembler Code for above Listing

```
                        XREF  i UseI                     EQU   i
Constant                EQU   1
Sum                     EQU   Constant + $1000 + @234
```

The hexadecimal C constant `0x1000` was translated to `$1000` while the octal `01234` was translated to `@1234`. In addition, the compiler has inserted one space between every two tokens. These are the only changes the compiler makes in the assembler listing for defines.

Macros with parameters, predefined macros, and macros with no defined value are not generated.

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: Assembler Code for Above Listing

```
                        XREF  i AddressOfI               EQU   & i
ConstantInt             EQU   ( ( int ) 1 )
useUndef                EQU   UndefFkt * 6
Anything                EQU   § § / % & % / & + * % ç 65467568756 86
```

The `AddressOfI` macro does not assemble because the assembler does not know to interpret the `& C` address operator. Also, other C-specific operators such as dereferenciation (`*ptr`) must not be used. The compiler generates them into the assembler listing file without any translation.

The `ConstantInt` macro does not work because the assembler does not know the cast syntax and the types.

Macros with parameters are not written to the listing. Therefore, `Mul7` does not occur in the listing. Also, macros defined as Nothing, with no actual value, are not generated.

The C preprocessor does not care about the syntactical content of the macro, though the assembler `EQU` directive does. Therefore, the compiler has no problems with the `useUndef` macro using the undefined object `UndefFkt`. The assembler `EQU` directive requires that all used objects are defined.

The Anything macro shows that the compiler does not care about the content of a macro. The assembler, of course, cannot treat these random characters.

These types of macros are in a header file used to generate the assembler include file. They must only be in a region started with `#pragma CREATE_ASM_LISTING OFF` so that the compiler will not generate anything for them.

10.9.5.3 enums

`enums` in C have a unique name and a defined value. They are simply generated by the compiler as an `EQU` directive.

Listing: enum

```
#pragma CREATE_ASM_LISTING ON
enum {

  E1=4,

  E2=47,

  E3=-1*7

};
```

The enum code in the following listing results in the following EQUs:

Listing: Resultant EQUs from enums

```
E1                        EQU $4
E2                        EQU $2F

E3                        EQU $FFFFFFF9
```

NOTE

Negative values are generated as 32-bit hex numbers.

10.9.5.4 Types

As it does not make sense to generate the size of any occurring type, only `typedefs` are considered.

The size of the newly defined type is specified for all `typedefs`. For the name of the size of a `typedef`, an additional term `_SIZE` is appended to the end of the name. For structures, the offset of all structure fields is generated relative to the structure's start. The names of the structure offsets are generated by appending the structure field's name after an underline (_) to the typedef's name.

Listing: typedef and struct

```
#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;
```

Creates:

Listing: Resultant EQUs

```
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
```

All structure fields inside of another structure are contained within that structure. The generated name contains all the names for all fields listed in the path. If any element of the path does not have a name (e.g., an anonymous union), this element is not generated.

The width and the offset are also generated for all bitfield members. The offset 0 specifies the least significant bit, which is accessed with a `0x1` mask. The offset 2 specifies the most significant bit, which is accessed with a `0x4` mask. 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 structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

10.9.5.5 Functions

Declared functions are generated by the `XREF` directive. This enables them to be used with the assembler. Do not generate the function to be called from C, but defined in assembler, into the output file as the assembler does not allow the redefinition of labels declared with `XREF`. Such function prototypes are placed in an area started with `#pragma CREATE_ASM_LISTING OFF`, as shown in the following listing.

Listing: Function prototypes

```
#pragma CREATE_ASM_LISTING ON
void main(void);

void f_C(int i, long l);

#pragma CREATE_ASM_LISTING OFF

void f_asm(void);
```

Creates:

Listing: Functions defined in assembler

```
    XREF  main
    XREF  f_C
```

10.9.5.6 Variables

Variables are declared with `XREF`. In addition, for structures, every field is defined with an `EQU` directive. For bitfields, the bit offset and bit size are also defined.

Variables in the `__SHORT_SEG` segment are defined with `XREF.B` to inform the assembler about the direct access. Fields in structures in `__SHORT_SEG` segments, are defined with a `EQU.B` directive.

Listing: struct and variable

```
#pragma CREATE_ASM_LISTING ON
struct A {
  char a;
  int i:2;
};
struct A VarA;
#pragma DATA_SEG __SHORT_SEG ShortSeg
int VarInt;
```

Creates:

Listing: Resultant XREFs and EQUs

```
                        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 variable size is not explicitly written. 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 structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

10.9.5.7 Comments

Comments inside a region generated with `#pragma CREATE_ASM_LISTING ON` are also written on a single line in the assembler include file.

Comments inside of a typedef, a structure, or a variable declaration are placed either before or after the declaration. They are never placed inside the declaration, even if the declaration contains multiple lines. Therefore, a comment after a structure field in a typedef is written before or after the whole typedef, not just after the type field. Every comment is on a single line. An empty comment (`/* */`) inserts an empty line into the created file.

See the following listing for an example of how C source code with its comments is converted into assembly.

Listing: C source code conversion to assembly

```
#pragma CREATE_ASM_LISTING ON
/*

   The main() function is called by the startup code.

   This 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

Creates:

; 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, its incorrect use must be avoided using the following guidelines implemented in a real project. This section describes how the programmer uses 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.

- All interface memory reservations or definitions must be made 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 files-dependent include files to be created.
- Use `#pragma CREATE_ASM_LISTING ON/OFF` only in C header files. This #pragma selects the objects to translate to the assembler include file. The created assembler include 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) compiler 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.
- Check if the compiler-created assembler include file is included into your assembler source.

NOTE

In case of a zero-page declared object (if this is supported by the target), the compiler translates it into an `XREF.B` directive for the base address of a variable or constant. The compiler translates structure fields in the zero page into an `EQU.B` directive in order to access them. Explicit zero-page addressing syntax may be necessary as some assemblers use extended addresses to `EQU.B` defined labels. Project-defined data types must be declared in the C header file by including a global project header (e.g., `global.h`). This is necessary as the header file is compiled in a standalone fashion.

Chapter 11
Generating Compact Code

The Compiler tries whenever possible to generate compact and efficient code. But not everything is handled directly by the Compiler. With a little help from the programmer, it is possible to reach denser code. Some Compiler options, or using `__SHORT_SEG` segments (if available), help to generate compact code.

11.1 Compiler Options

Using the following compiler options helps to reduce the size of the code generated. Note that not all options may be available for each target.

11.1.1 -Oi: Inline Functions

Use the inline keyword or the command line option -Oi for C/C++ functions. The following listing defines a function before it is used helps the Compiler to inline it:

Listing: Example - Defining a Function

```
/* OK */

void fun(void);

void main(void) {

   fun();

}

void fun(void) {

  // ...
```

```
/* better! */

void fun(void) {

  // ...

}

void main(void) {

    fun();

}
```

```
}
```

This also helps the compiler to use a relative branch instruction instead of an absolute branch instruction.

11.2 Relocatable Data

The limited RAM size of the RS08 requires careful data allocation. Access global read/write data using tiny, short, direct, or paged addressing. Access global read-only data using paged or far addressing. RS08 non-static local data must be allocated into an OVERLAP section. The compiler does this allocation automatically, using the same address range as is used for direct addressing (0x00 to 0xBF).

- Use tiny addressing (`__TINY_SEG`) for frequently accessed global variables when the operand can be encoded on four bits. The address range for these variables is 0x00 - 0x0F.
- Use short addressing (`__SHORT_SEG`) to access IO registers in the lower RS08 register bank, when the operand can be encoded on five bits. The address range for these variables is 0x00 - 0x1F.
- Use direct (8-bit) addressing (`DEFAULT`) to access global variables when the operand is greater than four or five bits. The address range for these variables is 0x00 to 0xBF.
- Use paged addressing (`__PAGED_SEG`) to access IO registers in the upper RS08 register bank. The address range for these variables is 0x100 to 0x3FF.
- Use paged addressing (`__PAGED_SEG`) to access global read-only (constant) data. Objects allocated in PAGED sections must not cross page boundaries. The address range for these constants is 0x00 to 0x3FFF.
- Use far addressing (`__FAR_SEG`) to access large constant data. Allocate far sections to more than one page. The address range for these constants is 0x00 to 0x3FFF.

See the following listing for examples using the different addressing modes.

Listing: Allocating variables on the RS08

```
/* in a header file */ #pragma DATA_SEG __TINY_SEG MyTinySection
char status;
#pragma DATA_SEG __ SHORT_SEG MyShortSection
unsigned char IOReg;
#pragma DATA_SEG DEFAULT
char temp;
#pragma DATA_SEG __ PAGED_SEG MyShortSection
unsigned char IOReg;
unsigned char *__paged io_ptr = &IOREG;
#pragma DATA_SEG __ PAGED_SEG MyPagedSection
const char table[10];
unsigned char *__paged tblptr = table;
#pragma DATA_SEG __ FAR_SEG MyFarSection
```

```
const char table[1000];
unsigned char *__far tblptr = table;
```

The segment must be placed on the direct page in the PRM file.

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
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.

11.2.1 Using -Ostk

When you use the `-Ostk` option, the compiler analyzes which local variables are alive simultaneously. Based on that analysis the compiler chooses the best memory layout for for variables. When using `-Ostk`, two or more variables may end up sharing the same memory location.

11.3 Programming Guidelines

Following a few programming guidelines helps to reduce code size. Many things are optimized by the Compiler. However, if the programming style is very complex or if it forces the Compiler to perform special code sequences, code efficiency is not would be expected from a typical optimization.

11.3.1 Constant Function at a Specific Address

Sometimes functions are placed at a specific address, but the sources or information regarding them are not available. The programmer knows that the function starts at address 0x1234 and wants to call it. Without having the definition of the function, the program runs into a linker error due to the lack of the target function code. The solution is to use a constant function pointer:

```
void (*const fktPtr)(void) = (void(*)(void))0x1234;

void main(void) {

  fktPtr();

}
```

This gives you efficient code and no linker errors. However, it is necessary that the function at `0x1234` really exists.

Even a better way (without the need for a function pointer):

```
#define erase ((void(*)(void))(0xfc06))

void main(void) {

  erase(); /* call function at address 0xfc06 */

}
```

11.3.2 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 foo(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  __asm {
    /* some HLI statements */
  }
  /* maybe other C/C++ statements */
}
```

The Compiler in the worst case has to assume that everything has changed. It cannot hold variables into registers over HLI statements. Normally it is better to place special HLI code sequences into separate functions. However, there is the drawback of an additional call or return. Placing HLI instructions into separate functions (and module) simplifies porting the software to another target.

Listing: HLI Statements are not mixed with C Statements (recommended).

```
/* hardware.c */
void special_hli(void) {
  __asm {
    /* some HLI statements */
  }
}
/* foo.c */
void foo(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  special_hli();
  /* maybe other C/C++ statements */
}
```

11.3.3 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 powerful postincrement or postdecrement and preincrement or predecrement instructions, it is not recommended to use the `++' and `--' operator in complex expressions. Especially postincrement or postdecrement may result in additional code:

```
a[i++] = b[--j];
```

Write the above statement as:

```
j--; a[i] = b[j]; i++;
```

Using 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:

11.3.3.1 Example

```
i = 3;
```

```
fun(i++, --i);
```

In the above example, `fun()` is called either with `fun(3,3)` or with `fun(2,2)`.

11.3.4 Boolean Types

In C, the boolean type of an expression is an `int'. A variable or expression evaluating to 0 (zero) is FALSE and everything else (!= 0) is TRUE. Instead of using an `int` (usually 16 or 32 bits), it may be better to use an 8-bit type to hold a boolean result. For ANSI-C compliance, the basic boolean types are declared in `stdtypes.h`:

```
typedef int Bool;

#define TRUE  1

#define FALSE 0
```

Using `typedef Byte Bool_8` from `stdtypes.h` (`Byte` is an unsigned 8-bit data type also declared in `stdtypes.h`) reduces memory usage and improves code density.

11.3.5 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 on how to save code (not using float or doubles in `printf` may result in half the code).

11.3.6 Bitfields

Using bitfields to save memory may be a bad idea as bitfields produce a lot of 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 could 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.

11.3.7 Struct Returns

Normally the compiler must first allocate space for the return value (1) and then to call the function (2). In phase (3) the return value is copied to the variable `s`. In the callee `fun`, during the return sequence, the Compiler must copy the return value (`4, struct copy`).

Depending on the size of the `struct`, this may be done inline. After return, the caller `main` must copy 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, and increase code size and memory usage.

Listing: Returning a struct can force the Compiler to produce lengthy code.

```
struct S fun(void)
  /* ... */
  return s; // (4)
}
void main(void) {
  struct S s;
  /* ... */
  s = fun();  // (1), (2), (3)
  /* ... */
}
```

With the example in the following listing, the Compiler just has to pass the destination address and to call `fun` (2). On the callee side, the callee copies the result indirectly into the destination (4). This approach reduces memory 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 (e.g., if the destination `struct` is modified in the callee).

Listing: A better way is to pass only a pointer to the callee for the return value.

```
void fun(struct S *sp) {
  /* ... */
  *sp = s; // (4)
}
void main(void) {
  S s;
  /* ... */
  fun(&s);  // (2)
  /* ... */
}
```

11.3.8 Local Variables

Using local variables instead of global variable results in better manageability of the application as side effects are reduced or totally avoided. Using local variables or parameters reduces global memory usage but increases local memory usage.

Memory access capabilities of the target influences the code quality. Depending on the target capabilities, access efficiency to local variables may vary. Allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the memory in the beginning of the function and to deallocate it in the end:

Listing: Good candidate for global variables

```
void fun(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. A solution is to use global or static local variables. This deteriorates maintainability and also may waste global address space.

The RS08 Compiler overlaps parameter or local variables using a technique called overlapping. The Compiler allocates local variables or parameters as global, and the linker overlaps them depending on their use. Since the RS08 has no stack, this is the only solution. However this solution makes the code non-reentrant (no recursion is allowed).

11.3.9 Parameter Passing

Avoid parameters which exceed the data passed through registers (see Backend).

11.3.10 Unsigned Data Types

Using unsigned data types is acceptable as signed operations are much more complex than unsigned ones (e.g., shifts, divisions and bitfield operations). But it is a bad idea to use unsigned types just because a value is always larger or equal to zero, and because the type can hold a larger positive number.

11.3.11 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++);
```

11.3.11.1 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++);
```

11.3.11.2 memcpy() and memcpy2()

ANSI-C requires that the `memcpy()` library function in `strings.h` returns a pointer of the destination and handles and is able to also 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);
}
```

If the function does not have to return the destination and it has to handle a count of zero, the `memcpy2()` function in the following listing is much simpler and faster:

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.

11.3.12 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.

11.3.13 Tiny or Short Segments

Whenever possible, place frequently used global variables into a `__TINY_SEG` or `__SHORT_SEG` segment using:

```
#pragma DATA_SEG __SHORT_SEG MySeg
```

or

```
#pragma DATA_SEG __TINY_SEG MySeg
```

11.3.14 Qualifiers

Use the `const` qualifier to help the compiler. The `const` objects are placed into ROM for the Freescale object-file format if the `-Cc` compiler option is given.

Chapter 12
RS08 Backend

The Backend is the target-dependent part of a Compiler, containing the code generator. This section discusses the technical details of the Backend for the RS08 family.

12.1 Non-ANSI Keywords

The following table gives an overview about the supported non-ANSI keywords:

Table 12-1. Supported Non-ANSI Keywords

Keyword	Data Pointer	Supported for Function Pointer	Function
__far	yes	no	no
__near	yes	no	no
interrupt	no	no	yes
__paged	yes	no	no

12.2 Memory Models

This section describes the following memory models:

- SMALL Memory Model
- BANKED Memory Model

12.2.1 SMALL Memory Model

This is the default memory model for the RS08 compiler. It uses 8-bit accesses for data and 16-bit accesses (extended addressing) for functions.

12.2.2 BANKED Memory Model

The RS08 core does not support extended addressing for data access beyond the first 256 bytes. Therefore, a paging scheme has been implemented, that segments the full 16-Kbyte address map into 256 pages of 64 bytes each. The BANKED memory model is identical to the SMALL memory model in terms of function access, but uses this paging scheme to access data. The compiler accesses all data as `__paged` (that is, using 16-bit accesses with the page number in the high byte, and the offset in the low byte). This paging mechanism no longer works if the object crosses page boundaries. `__far` addressing is more expensive (the page register must be updated for each byte access).

12.2.2.1 See also

- Pointer Qualifiers

12.3 Data Types

This section describes how the basic types of ANSI-C are implemented by the RS08 Backend.

12.3.1 Scalar Types

All basic types may be changed with the -T option. Note that all scalar types (except char) have no signed/unsigned qualifier, and are considered signed by default, for example `int` is the same as `signed int`.

The sizes of the simple types are given by the table below together with the possible formats using the `-T` option:

Table 12-2. Scalar Types

Type	Default Format	Default Value Range	Default Value Range	Formats Available With Option -T
		Min	Max	
char (unsigned)	8bit	0	255	8bit, 16bit, 32bit
singned char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit
signed short	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned short	16bit	0	65535	8bit, 16bit, 32bit
enum (signed)	16bit	-32768	32767	8bit, 16bit, 32bit
signed int	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned int	16bit	0	65535	8bit, 16bit, 32bit
signed long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long	32bit	0	4294967295	8bit, 16bit, 32bit
signed long long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long long	32bit	0	4294967295	8bit, 16bit, 32bit

NOTE

Plain type `char` is unsigned. This default is changed with the `-T` option.

12.3.2 Floating Point Types

The RS08 compiler supports IEEE32 floating point calculations. The compiler uses IEEE32 format for both types.

The option `-T` may be used to change the default format of float/double.

Table 12-3. Floating Point Types

Type	Default Format	Default Value Range	Default Value Range	Formats Available With Option -T
		Min	Max	
float	IEEE32	-1.17549435E-38F	3.402823466E+38F	IEEE32
double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
long long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32

12.3.3 Pointer Types and Function Pointers

The following table shows how data pointer size varies with memory model and `__far`, `__near`, or `__paged` keyword usage.

Table 12-4. Data Pointer Sizes

Memory Model	Compiler Option	Default Pointer Size	`__near` Pointer Size	`__far` Pointer Size	`__paged` Pointer Size
SMALL	`-Ms`	1	1	2	2
BANKED	`-Mb`	2	1	2	2

Function pointer size is 2 bytes, regardless of the memory model selected.

12.3.4 Structured Types, Alignment

Local variables are allocated in overlapping areas. The most significant part of a simple variable is stored at the low memory address.

12.3.5 Bitfields

The maximum width of bitfields is 32 bits. The allocation unit is one byte. The Compiler uses words only if a bitfield is wider than eight bits. Allocation order is from the least significant bit up to the most significant bit in the order of declaration.

12.4 Register Usage

The RS08 Compiler and library use five pseudo registers defined in the table below:

Table 12-5. Pseudo Registers

Register	Normal Function
U (or _U)	scratch register
V (or _V)	scratch register
W (or _W)	scratch register
Y (or _Y)	address register of binary operations
Z (or _Z)	scratch register

12.5 Parameter Passing

These parameter passing and return value conventions apply for the RS08:

- Registers used for Parameter Passing (only for parameter lists without the ellipsis):
  - if the last parameter is 8-bit large, the parameter is passed in register A
  - all other are parameters passed using the shared section (`OVERLAP`)
- Parameter naming conventions for the shared `OVERLAP` section are zero-relative starting from the last parameter. In other words:
  - `__OVL_funcname_p0` indicates the last parameter
  - `__OVL_funcname_p1` indicates the next to the last parameter
  - `__OVL_funcname_p2` indicates the third from the last parameter, and so on.
- Ellipsis (for open parameters): Pass open parameters using a parameter block allocated in the local scope of the caller. The address of this block is passed implicitly together with the last declared parameter. The callee uses this address to get the open parameters:

```
unsigned char __OVL_callerfuncname_pblock [MAX_BLOCK_SIZE] // MAX_BLOCK_SIZE is the
maximum of bytes needed by a caller to pass open parameters
```

- The following return address is used for non-leaf functions or for functions that call the runtime library):

```
unsigned char __OVL_funcname_ra[2] // 2 bytes for return address of caller
```

12.6 Entry and Exit Code

The RS08 uses an overlapping system, rather than a stack, for memory allocation. The following listing illustrates correct entry and exit code use.

Listing: Entry and Exit Code Example

```
10: int f(int x) {

0000 45 SHA

0001 b700 STA __OVL_f_14__PSID_75300003 ------> entry code. Saves the
SPC value. This only happens for non-leaf functions !

0003 42 SLA
0004 b701 STA __OVL_f_14__PSID_75300003:1

11: g(&x);

0006 a600 LDA #__OVL_f_p0

0008 ad00 BSR PART_0_7(g)

12: return x;

000a 4e000f LDX __OVL_f_p1 --------> stores the return value

000d 4e000e MOV __OVL_f_p0,D[X]

0010 2f INCX

0011 4e010e MOV __OVL_f_p0:1,D[X]

0014 b600 LDA __OVL_f_14__PSID_75300003 -----> restore SPC. This only
happens for non-leaf functions !

0016 45 SHA

0017 b601 LDA __OVL_f_14__PSID_75300003:1

0019 42 SLA

13: }

001a be RTS
```

12.7 Pragmas

The Compiler provides some pragmas that control the generation of entry and exit code.

12.7.1 TRAP_PROC

The return instruction from the function is `JMP IEA` (jump to the Interrupt Exit Address register). At this address, a jump opcode is hard-wired, and the destination of the jump is stored by the core in the Interrupt Return Address register (IRA). When using `#pragmaTRAP_PROC`, the user can configure which registers should be saved when entering the interrupt handler and restored at exit.

The syntax for this pragma is:

```
#pragma TRAP_PROC [SAVE_A|SAVE_X|SAVE_PAGE].
```

NOTE

The arguments can be provided in any combination, for example `#pragma TRAP_PROC SAVE_A SAVE_PAGE`, and have as effect saving the accumulator register, the index register, and the PAGESEL register respectively.

12.7.2 NO_ENTRY

Omits generation of procedure entry code.

12.7.3 NO_EXIT

Does not generate procedure exit code.

12.8 Interrupt Functions

Interrupt procedures are marked with the interrupt keyword. These functions return with a JMP IEA instruction instead of RTS and should not have any arguments.

12.8.1 Interrupt Vector Table Allocation

The vector table has to be setup with normal C (or assembly) code.

Instead an array of vectors has to be allocated and initialized with the address of the handlers and with their initial thread argument.

12.9 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 then are allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

There are two basic types of segments, code and data segments, each with a matching pragma:

```
#pragma CODE_SEG  <name>

#pragma DATA_SEG  <name>
```

Both are valid until the next pragma of the same kind is encountered. If no segment is specified, the Compiler assumes two default segments named `DEFAULT_ROM` (the default code segment) and `DEFAULT_RAM` (the default data segment). To explicitly make these default segments the current ones, use the segment name `DEFAULT`:

```
#pragma CODE_SEG DEFAULT

#pragma DATA_SEG DEFAULT
```

12.10 Optimizations

The Compiler applies a variety of code improving techniques commonly called optimizations. This section gives a short overview about the most important optimizations.

12.10.1 Lazy Instruction Selection

Lazy instruction selection is a very simple optimization that replaces certain instructions by shorter and/or faster equivalents. Examples are the use of `TSTA` instead of `CMPA #0` or using `COMB` instead of `EORB #0xFF`.

12.10.2 Branch Optimizations

The Compiler uses branch instructions with short offsets whenever possible. Additionally, other optimizations for branches are also available.

12.10.3 Constant Folding

Constant folding options only affect constant folding over statements. The constant folding inside of expressions is always done.

12.10.4 Volatile Objects

The Compiler does not do register tracing on volatile objects. Accesses to volatile objects are not eliminated. It also doesn't change word operations to byte operations on volatile objects as it does for other memory accesses.

12.11 Programming Hints

The RS08 is an 8-bit processor not designed with high-level languages in mind. You must observe certain points to allow the Compiler to generate reasonably efficient code. The following list provides an idea of what is "good" programming from the processor's point of view.

- Use the restrict keyword as a hint for the pointer to thread function argument descriptors.
- Use 8-bit computations unless larger types are absolutely required. The RS08 core is an 8-bit MCU, so 16-bit arithmetic operations are expensive, since they are implemented by runtime calls.
- Limit the number of local variables, since storage space is limited.

Using `unsigned` types instead of `signed` types is better in the following cases:

- Implicit or explicit extensions from `char` to `int` or from `int` to `long`.
- Use types `long`, `float` or `double` only when absolutely necessary. They produce a lot of code.

Chapter 13
High-Level Inline Assembler for the Freescale RS08

The HLI (High Level Inline) Assembler provides a means to make full use of the properties of the target processor 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/C++ with the inline assembler. The Compiler does all that work for you. For more information, refer to the RS08 Reference Manual.

13.1 Syntax

Inline 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, shown in various configurations: `__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` If you use the first form, multiple __ `asm` statements are contained on one line and comments are delimited like regular C or C++ comments. If you use the second form, one to several assembly instructions are contained within the __ `asm` block, but only one assembly instruction per line is possible and the semicolon starts an assembly comment.

13.1.1 Mixing HLI Assembly and HLL

Mixing High Level Inline (HLI) Assembly with a High Level Language (HLL, for example C or C++) requires special attention. The Compiler does care about used or modified registers in HLI Assembly, thus you do not have save/restore registers which are used in HLI. It is recommended to place complex HLI Assembly code, or HLI Assembly code modifying any registers, into separate functions.

Example:

```
void foo(void) {
  /* some C statements */
  p->v = 1;
  __asm {
    /* some HLI statements destroying registers */
  }
  /* some C statements */
  p->v = 2;
}
```

In the above sequence, the Compiler holds the value of `p` in a register. The compiler will correctly reload `p` if necessary.

13.1.2 Example

A simple example illustrates the use of the HLI-Assembler. Assume the following:

- `from` points to some memory area
- `to` points to some other, non-overlapping memory area.

Then we can write a simple string copying function in assembly language as follows:

```
void _CMPS(void) {
  __asm {
       ADD         #128
       STA         _Z
       LDA         _Y
       ADD         #128
       CMP         _Z
  }
}
```

13.1.3 C Macros

The C macros are expanded inside of inline assembler code as they are expanded in C. One special point to note is the syntax of a `__asm` directive generated by macros. As macros always expand to one single line, only the first form of the `__asm` keyword is used in macros:

```
__asm NOP;
```

For example:

```
#define SPACE_OK { __asm NOP; __asm NOP; }
```

Using the second form is invalid:

```
#define NOT_OK { __asm { \
                                           NOP; \
                                           NOP; \
                                                }
```

The macro `NOT_OK` is expanded by the preprocessor to one single line, which is then incorrectly translated because every assembly instruction must be explicitly terminated by a `newline`. Use the pragma `NO_STRING_CONSTR` to build immediates by using `#` inside macros.

13.2 Special Features

The following special features are available with the RS08 compiler.

13.2.1 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.

13.2.2 Reserved Words

The inline assembler knows a couple of reserved words, which must not collide with user defined identifiers such as variable names. These reserved words are:

- All opcodes `(MOV, NOP, ...)`
- All register names `(A, X, D[X])`
- The fixup identifiers:

Table 13-1. Fixup Identifiers

Name	Address Kind	Description
`%HIGH_6_13`	Logical Address	Returns the page number corresponding to a given RS08 14-bit address.
`%MAP_ADDR_6`	Logical Address	Returns the offset within the paging window corresponding to a given RS08 address.

For these reserved words, the inline assembler is *not* case sensitive, that is `JSR` is the same as `jsr` or even `JsR`. For all other identifiers (labels, variable names and so on) the inline assembler is case sensitive. The following example shows the syntax of the fixup specification:

```
  __asm MOV   #%HIGH_6_13(var),__PAGESEL
```

13.2.3 Pseudo-Opcodes

This ection lists the pseudo opcodes for inline assembler.

13.2.3.1 DC - Define constant

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

DC.B 1 ; Byte constant 1

DC.B 0 ; Byte constant 0

DC.W 12 ; Word constant 12

DC.L 20,23 ; Longword constants

13.2.3.2 __RETADDR - Specify the return address location

Given a non-leaf function, that is, one that contains JSR/BSR instructions, the value of the Shadow PC (SPC) register must be saved and restored accordingly. The value is stored to a temporary in the compiler-generated entry code, and subsequently restored to the register in the exit code, which is also generated by the compiler. If, however, the function contains only inline assembly and has #pragma NO_ENTRY applied to it, even though the SPC value has been properly manipulated, this time by the programmer, the generated debug information might be incomplete (the debugger might not be able to recreate the stack trace) - unless the __RETADDR pseudo-opcode is used to specify in which variable has the return address been stored. The following listing contains an example of using __RETADDR

Listing: Using __RETADDR to specify the return address location

```
#pragma NO_ENTRY

#pragma NO_EXIT

void foo(void) {

  unsigned int ret;

  asm {

    STA  tmp

    SHA

    STA  ret

    SLA

    STA  ret+1

    __RETADDR ret  /* starting from this PC value on, the return
address resides in variable 'ret' */

    LDA  tmp

    ...

  }

}
```

13.2.4 Accessing Variables

The inline assembler allows accessing local and global variables declared in C by using their name in the instruction. For global variable names, use the correct fixup specification (usually `%LOWC` for the low byte and `%HIGH` for the high byte part).

13.2.5 Constant Expressions

Constant expressions may be used anywhere an *IMMEDIATE* value is expected. The HLI supports the same operators as in ANSI-C code. The syntax of numbers is the same as in ANSI-C.

Chapter 14
ANSI-C Library Reference

This section covers the ANSI-C Library.

- Library Files : Description of the types of library files
- Special Features : Description of special considerations of the ANSI-C standard library relating to embedded systems programming
- Library Structure : Examination of the various elements of the ANSI-C library, grouped by category.
- Types and Macros in Standard Library : Discussion of all types and macros defined in the ANSI-C standard library.
- Standard Functions : Description of all functions in the ANSI-C library

Chapter 15
Library Files

The chapter describes library files.

15.1 Directory Structure

The library files are delivered in the following structure:

Listing: Layout of files after a CodeWarrior installation/

```
CWInstallDir\MCU\lib\<target>c\               /* readme files, make
files */
CWInstallDir\MCU\lib\<target>c\src            /* C library source
files   */

CWInstallDir\MCU\lib\<target>c\include        /* library include
files    */

CWInstallDir\MCU\lib\<target>c\lib            /* default library
files    */

CWInstallDir\MCU\lib\<target>c\prm            /* Linker parameter
files   */
```

Read the `README.TXT` located in the library folder for additional information on memory models and library filenames.

NOTE

The RS08 and the HC08 share the standard library files. Therefore, the RS08 library files are located in, *CWInstallDir* \`MCU\lib\hc08c` *where, CWInstallDir* is the directory in which the CodeWarrior software is installed.

15.2 Generating a Library

In the directory structure above, a CodeWarrior `*.mcp` file is provided to build all the libraries and the startup code object files. Simply load the `<target>_lib.mcp` file into the CodeWarrior IDE and build all the targets.

15.3 Common Source Files

The following table lists the source and header files of the Standard ANSI Library that are not target-dependent.

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, there are also startup object files compiled with different Compiler option settings (see Compiler options for details).

The correct startup file must be linked with the application depending on the memory model chosen. 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
- 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, the version written in C could be used as well. For example, just compile the ` `startup.c`' file with the memory/options settings and link it to the application.

NOTE

When using the small memory model, the size of data pointers is (if not specified otherwise) 8 bits. This means that any memory location above `0xFF` cannot be accessed using such a pointer. The startup code makes use of pointers when performing the zero-out and copy-down operations. If the application has data above the `0xFF` boundary, then the startup code has to use the `__far` pointers. To enforce this, the user has to provide `-D__STARTUP_USE_FAR_POINTERS` in the command line when compiling the startup code. This is also true when using the banked memory model, since in this situation the pointers are `__paged` by default. This means that in a scenario where the pointed data crossing page boundary is not supported, the startup code does not work with the `__paged` pointers.

15.5 Library Files

Most of the object files of the ANSI library are delivered in the form of an object library.

Several Library files are bundled with the Compiler. The reasons for having different library files are due to different memory models or floating point formats.

The library files contain all necessary runtime functions used by the compiler and the ANSI Standard Library as well. The list files (`*.lst` extension) contains a summary of all objects in the library file.

To link against a modified file which also exists in the library, it must be specified first in the link order.

Check out the `readme.txt` located in the library structure (`lib\<target>c\README.TXT`) for a list of all delivered library files and memory model or options used.

Chapter 16
Special Features

Not everything defined in the ANSI standard library makes sense in embedded systems programming. Therefore, not all functions have been implemented, and some have been left open to be implemented because they strongly depend on the actual setup of the target system.

This chapter describes and explains these points.

NOTE

All functions not implemented 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. Another function using a global variable is strtok(), because it has been defined that way in the ANSI standard.

16.1 Memory Management -- malloc(), free(), calloc(), realloc(); alloc.c, and heap.c

File `alloc.c` provides a full implementation of these functions. The only problems remaining are the question of heap location, heap size, and what happens when heap memory runs out.

All these points can be addressed in the `heap.c` file. The heap is viewed as a large array, and there is a default error handling function. Modify this function or the size of the heap to suit the needs of the application. The size of the heap is defined in `libdefs.h`, `LIBDEF_HEAPSIZE`.

16.2 Signals - signal.c

Signals have been implemented as traps. This means the signal() function allows you to set a vector to some function of your own (ideally a `TRAP_PROC`), while the raise() function is unimplemented. If you decide to ignore a certain signal, a default handler is installed that does nothing.

16.3 Multi-byte Characters - mblen(), mbtowc(), wctomb(), mbstowcs(), wcstombs(); stdlib.c

Because the compiler does not support multi-byte characters, all routines in `stdlib.c` dealing with those are unimplemented. If these functions are needed, the programmer must 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 it is found in the `terminal.c` file.

This difference has been planned because often no terminal is available at all or a terminal depends highly on the user hardware.

The ANSI library contains several functions which makes it simple to implement the `printf()` function with all its special cases in a few lines.

The first, ANSI-compliant way is to allocate a buffer and then use the `vsprintf()` ANSI function .

Listing: An 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()`. The `WriteString()` function is assumed to write one string to a terminal. There are two disadvantages to this solution:

- A buffer is needed which alone may use a large amount of RAM.
- As unimportant how large the buffer (`MAXLINE`) is, it is always possible that a buffer overflow occurs. Therefore this solution is not safe.

Two non-ANSI functions, `vprintf()` and `set_printf()`, are `provided` in its newer library versions in order to avoid both disadvantages. Because these functions are a non-ANSI extension, they are not contained in the `stdio.h` header file. Therefore, their prototypes must be specified before they are used:

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 which should be printed by `vprintf()`.

Be advised that the standard ANSI C `printf()` derivatives functions, sprintf() and `vsprintf()`, are also implemented by calls to `set_printf()` and `vprintf()`. This way much of the code for all printf derivatives can be shared across them.

There is also a limitation of the current implementation of printf(). Because the callback function is not passed as an argument to `vprintf()`, but held in a global variable, all the `printf()` derivatives are not reentrant. Even calls to different derivatives at the same time are not allowed.

For example, a simple implementation of a `printf()` with `vprintf()` and `set_printf()` is shown in the following listing:

Listing: Implementation of prinft() 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;
}
```

The `PutChar()` function is assumed to print one character to the terminal.

Another remark has to be made about the `printf()` and scanf() functions. The full source code is provided of all `printf()` derivatives in `"printf.c"` and of `scanf()` in `scanf.c`. Usually many of the features of `printf()` and `scanf()` are not used by a specific application. The source code of the library modules printf and scanf contains switches (defines) to allow the use to switch off unused parts of the code. This especially includes the large floating-point parts of `vprintf()` and `vsscanf()`.

16.6 Locales - locale.*

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 accesses to a lookup table.

This table uses 257 bytes of memory, so an implementation using real functions is provided. These are accessible if the macros are undefined first. After `#undef isupper`, `isupper` is translated into a call to function `isupper()`. Without the `undef`, `isupper` is replaced by the corresponding macro.

Using the functions instead of the macros of course saves RAM and code size - at the expense of some additional function call overhead.

16.8 String Conversions - strtol(), strtoul(), strtod(), and stdlib.c

To follow the ANSI requirements for string conversions, range checking has to be done. The variable `errno` is set accordingly and special limit values are returned. The macro `ENABLE_OVERFLOW_CHECK` is set to 1 by default. To reduce code size, switching this macro off is recommended (clear `ENABLE_OVERFLOW_CHECK` to 0).

Chapter 17
Library Structure

In this chapter, the various parts of the ANSI-C standard library are examined, grouped by category. This library not only contains a rich set of functions, but also numerous types and macros.

17.1 Error Handling

Error handling in the ANSI library is done using a global variable `errno` that is set by the library routines and may be tested by a user program. There also are a few functions for error handling:

Listing: Error handling functions

```
void  assert(int expr);
void  perror(const char *msg);

char * strerror(int errno);
```

17.2 String Handling Functions

Strings in ANSI-C always are 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. The main difference to the string functions is that they operate on any block of memory, whether it is null-terminated or not. The length of the block must be given as an additional parameter. Also, these functions work with `void` pointers instead of `char` pointers.

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 contains a variety of floating point functions. The standard interface, which is defined for type `double`, has been augmented by an alternate interface (and implementation) using type `float`.

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 functions using the `float` type have the same names with an `f` appended.

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 a couple of functions operating on integral values:

Listing: ANSI-C Integral functions

```
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 and a function for generating 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:

Listing: Memory allocation 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.cfile`, 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 searching and a generalized sorting procedure:

Listing: Generalized searching and sorting functions

```
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);

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 multibyte and wide characters. The implementation only offers minimum support for this feature: the maximum length of a multibyte character is one byte.

Listing: Interface for multibyte and wide characters

```
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);
```

17.7 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 functions shown in the following listing, implemented as macros:

Listing: Macros with variable-length arguments

```
void va_start(va_list args, param);
type va_arg(va_list args, type);
```

```
void va_end(va_list args);
```

17.8 Time Functions

In the ANSI library, there also are several function 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);
```

17.9 Locale Functions

These functions are for handling locales. The ANSI-C library only supports the minimal `C` environment.

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);
```

17.10 Conversion Functions

Functions for converting strings to numbers are found in the following listing.

Listing: AN SI-C string/number conversion functions

```
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);
```

17.11 printf() and scanf()

More conversions are possible for the C functions for reading and writing formatted data. These functions are shown in the following listing.

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, ...);
```

17.12 File I/O

The ANSI-C library contains a fairly large interface for file I/O. In microcontroller applications however, one usually does not need file I/O. In the few cases where one would need it, the implementation depends on the actual setup of the target system. Therefore, it is impossible for Freescale to provide an implementation for these features that the user has to specifically implement.

The following listing contains file I/O functions while **Listing: ANSI-C functions for writing and reading characters** has functions for the reading and writing of characters. The functions for reading and writing blocks of data are found in **Listing: ANSI-C**

functions for reading and writing blocks of data. Functions for formatted I/O on files are found in **Listing: ANSI-C formatted I/O functions on files**, and **Listing: ANSI-C positioning functions** has 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(
```

Chapter 18
Types and Macros in Standard Library

This chapter discusses all types and macros defined in the ANSI standard library. We cover each of the header files, in alphabetical order.

18.1 errno.h

This header file just declared two constants, that are used as error indicators in the global variable errno.

```
extern int errno;

 #define EDOM -1

 #define ERANGE -2
```

18.2 float.h

Defines constants describing the properties of floating point arithmetic. Refer to 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 a couple of constants for the maximum and minimum values that are allowed for certain types. Refer to 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 multi-byte character.
`SHRT_MIN`	Minimum value for short int
`SHRT_MAX`	Maximum value for short int
`USHRT_MAX`	Maximum value for unsigned short int
`INT_MIN`	Minimum value for int
`INT_MAX`	Maximum value for int
`UINT_MAX`	Maximum value for unsigned int

Table continues on the next page...

Table 18-3. Constants Defined in limits.h (continued)

Constant	Description
`LONG_MIN`	Minimum value for long int
`LONG_MAX`	Maximum value for long int
`ULONG_MAX`	Maximum value for unsigned long int

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. */
  har  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 */
};
```

There also are several constants that can be used in setlocale() to define which part of the locale to set. Refer to the following table:

Table 18-4. Constants used with setlocal()

Constant	Description
LC_ALL	Changes the complete locale
LC_COLLATE	Only changes the locale for the strcoll() and strxfrm() functions
LC_MONETARY	Changes the locale for formatting monetary numbers
LC_NUMERIC	Changes the locale for numeric, i.e., non-monetary formatting
LC_TIME	Changes the locale for the strftime() function
LC_TYPE	Changes the locale for character handling and multi-byte character functions

This implementation only supports the minimum C locale.

18.5 math.h

Defines just this constant:

```
HUGE_VAL
```

Large value that is returned if overflow occurs.

18.6 setjmp.h

Contains just 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. Refer to 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 could not be installed.
`SIG_IGN`	If passed as the second argument to signal(), the signal is ignored.

Table 18-6. Signal Type Constants

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 to 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

There are two type declarations in this header file. Refer to 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.
`SEEK_END`	fseek() positions from the end of the file.L
`SEEK_SET`	fseek() positions from the start of the file.
`TMP_MAX`	Maximum number of unique filenames tmpnam() can generate.

In addition, 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. See 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 file `string.h` defines only functions and not types or special defines.

The functions are explained below together with all other ANSI functions.

18.13 assert.h

The file `assert.h` defines the assert() macro. If the `NDEBUG` macro is defined, then assert does nothing. Otherwise, assert calls the auxiliary function _assert if the one macro parameter of assert evaluates to `0 (FALSE)`. See 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 file `stdarg.h` defines the type `va_list` and the va_arg(), va_end(), and va_start() macros. The `va_list` type implements a pointer to one argument of a open parameter list. The `va_start()` macro 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 `va_arg()` macro 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, which is accessed only with `sizeof()`. So type, but also variables can be used. See the following listing for an example using `stdarg.h`.

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, for integral types at least an int is passed and for floating types at least a double is passed. In other words, the result of using `va_arg(..., char)` or `va_arg(..., short)` is undefined in C. Be especially careful when using variables instead of types for `va_arg()`. In the example above, `res= va_arg(list, res)` is not correct unless `res` has the 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()`, and many others. These functions are either implemented as macros, or as real functions. The macro version is used when the -Ot compiler option is used or 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)     /* Uppercase         */
```

```
#define  _L  (1<<1)     /* Lowercase         */
#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
Standard Functions

This section describes all the standard functions in the ANSI-C library. Each function description contains the subsections listed in the following table.

Table 19-1. Function Description Subsections

Subsection	Description
Syntax	Shows the function's prototype and also which header file to include.
Description	A description of how to use the function.
Return	Describes what the function returns in which case. If the global variable `errno` is modified by the function, possible values are also described.
See also	Contains cross-references to related functions.

The functions listed in this section also include some functions which are not implemented in the Compiler. These unimplemented functions are categorized as:

- Hardware specific
- File I/O

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 (e.g., because it does a longjmp()), the application is not halted.

See also

`atexit()`,

`exit()`,

`raise()`, and

`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`; i.e., `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-32768`, this value is returned and `errno` is set to `ERANGE`.

See also

`fabs() and fabsf()`

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, `NAN` is returned and `errno` is set to `EDOM`.

See also

asin() and asinf(),

atan() and atanf(),

atan2() and atan2f(),

cos() and cosf(),

sin() and sinf(), and

tan() and tanf()

19.4 asctime()

This is a *Hardware-specific* function. It is not implemented in the 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()`, and

`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, `NAN` is returned and `errno` is set to `EDOM`.

See also

acos() and acosf(),

atan() and atanf(),

atan2() and atan2f(),

`cos()` and `cosf()`, and

tan() and tanf()

19.6 assert()

Syntax

```
#include <assert.h >
```

```
void assert(int expr);
```

Description

`assert()` is a macro that indicates 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.

See also

abort() and

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` radian

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` radian, if not both `x` and `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()`, and

`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()`. These functions are called in the reverse order they were registered.

Return

`atexit()` returns 0 if it could register the function, otherwise it returns a non-zero value.

See also

`abort()` and

`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, skipping over 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 the following:

```
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(), and

strtoul()

19.11 atoi()

Syntax

```
#include <stdlib.h >
```

```
int atoi(const char *s);
```

Description

`atoi()` converts the string `s` to an integer value, skipping over 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 the following:

```
Number              = [+|-]Digit{Digit}
```

Return

`atoi()` returns the converted integer value.

See also

atof(),

atol(),

strtod(),

strtol(), and

strtoul()

19.12 atol()

Syntax

```
  #include <stdlib.h >



  long atol(const char *s);
```

Description

`atol()` converts the string `s` to an `long` value, skipping over 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 atol() is the following:

```
Number = [+|-]Digit{Digit}
```

Return

`atol()` returns the converted `long` value.

See also

atoi(),

atof(),

strtod(),

strtol(), and

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 type `cmp_func` can be declared as:

```
typedef int (*cmp_func)(const void *key,
const void *data);
```

The comparison function returns an integer according to, as listed in the following table:

Table 19-2. Return value from the comparison function, cmp_func()

Key element value	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 arguments of `bsearch()` are as listed in the following table:

Table 19-3. Possible arguments to the bsearch() function

Parameter Name	Meaning
`key`	A pointer to the key data you are seeking
`array`	A pointer to the beginning (i.e., 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 element of the array that matches the key, if there is one. If the comparison function never returns zero, i.e., there is no matching array element, `bsearch()` returns `NULL`.

19.14 calloc()

This is a *Hardware-specific* function. It is not implemented in the 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`. All bytes in the memory block are initialized to zero. To deallocate the block, use free(). Do not use the default implementation in interrupt routines because it is not reentrant.

Return

`calloc()` returns a pointer to the allocated memory block. If the block cannot be allocated, the return value is `NULL`.

See also

`malloc()` and

`realloc()`

19.15 ceil() and ceilf()

Syntax

```
  #include <math.h >



  double ceil(double x);


float ceilf(float x);
```

Description

`ceil()` returns the smallest integral number larger than `x`.

See also

floor() and floorf() and

fmod() and fmodf()

19.16 clearerr()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h>

void clearerr(FILE *f);
```

Description

`clearerr()` resets the error flag and the `EOF` marker of file `f`.

19.17 clock()

This is a *Hardware-specific* function. It is not implemented in the 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(), and

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, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

cos() and cosf(),

sinh() and sinhf(), and

tanh() and tanhf()

19.20 ctime()

This is a *Hardware-specific* function. It is not implemented in the 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.

See also

`asctime()`,

`mktime()`, and

`time()`

19.21 difftime()

This is a *Hardware-specific* function. It is not implemented in the 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()` and

`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`

The `status` argument is ignored.

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, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

`log() and logf()`,

`log10() and log10f()`, and

`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() and

labs()

19.26 fclose()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >



  int fclose(FILE *f);
```

Description

`fclose()` closes file f. Before doing so, it 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, i.e., not using setbuf().

Return

Zero, if the function succeeds; `EOF` otherwise.

See also

fopen()

19.27 feof()

This is a *File I/O* function. It is not implemented in the 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 you are not at the end of the file; `EOF` otherwise.

19.28 ferror()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >



  int ferror(FILE *f);
```

Description

`ferror()` tests whether an error had occurred on file `f`. To clear the error indicator of a file, 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 there was no error; non-zero otherwise.

19.29 fflush()

This is a *File I/O* function. It is not implemented in the 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`, *all* files open for writing are flushed.

Return

Zero, if there was no error; `EOF` otherwise.

See also

setbuf() and

setvbuf()

19.30 fgetc()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >


int fgetc(FILE *f);
```

Description

`fgetc()` reads the next character from file `f`.

NOTE

If file `f` had been opened as a text file, the end-of-line character combination is read as one `'\n'` character.

Return

The character is read as an integer in the range from 0 to 255. If there was a read error, `fgetc()` returns `EOF` and sets the file's error flag, so that a subsequent call to ferror() will return 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() will return `EOF`, but ferror() will return `0`.

See also

fgets(),

fopen(),

fread(),

fscanf(), and

getc()

19.31 fgetpos()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >



int fgetpos(FILE *f, fpos_t *pos);
```

Description

`fgetpos()` returns the current file position in `*pos`. This value can be used to later set the position to this one 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 there is a `fsetpos()` with that value resets the position in the file correctly.

Return

Non-zero, if there was an error; zero otherwise.

See also

fseek() and

ftell()

19.32 fgets()

This is a *File I/O* function. It is not implemented in the 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, a `'\0'` is appended. 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 there was an error; `s` otherwise.

See also

fgetc() and

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 not larger than `x`.

Return

The largest integral number not larger than `x`.

See also

ceil() and ceilf() and

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`.

See also

div(),

ldiv(),

ldexp() and ldexpf(), and

modf() and modff()

19.35 fopen()

This is a *File I/O* function. It is not implemented in the 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 (i.e., 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 19-4. Operating modes of the file opening function, fopen()

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.

Table continues on the next page...

Table 19-4. Operating modes of the file opening function, fopen() (continued)

Mode	Effect
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, appending all writes to the end.

If the mode contains an " `r`", but the file does not 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 the file could not be created, the function returns `NULL`.

See also

fclose(),

freopen(),

setbuf() and

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 occurs, `fprintf()` returns `EOF`.

See also

printf() and

vfprintf(), vprintf(), and vsprintf()

19.37 fputc()

This is a *File I/O* function. It is not implemented in the 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 occurs, `fputc()` returns `EOF`.

See also

fputs()

19.38 fputs()

This is a *File I/O* function. It is not implemented in the 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 there was an error; zero otherwise.

See also

fputc()

19.39 fread()

This is a *File I/O* function. It is not implemented in the 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(), and

fwrite()

19.40 free()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <stdlib.h >

void free(void *ptr);
```

Description

`free()` deallocates a memory block that had previously been allocated by calloc(), malloc(), or realloc(). If `ptr` is `NULL`, nothing happens. Do not use the default implementation in interrupt routines because it is not reentrant.

19.41 freopen()

This is a *File I/O* function. It is not implemented in the 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. This can be useful for redirecting `stdin`, `stdout`, or `stderr`. About possible modes, see fopen().

See also

`fclose()`

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 is normalized 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(), and

modf() and modff()

19.43 fscanf()

This is a *File I/O* function. It is not implemented in the 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(), and

scanf()

19.44 fseek()

This is a *File I/O* function. It is not implemented in the 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 the following table.

Table 19-5. Offset position into the file for the fseek() function

Mode	Offset 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(), and

fsetpos()

19.45 fsetpos()

This is a *File I/O* function. It is not implemented in the 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 it was successful; non-zero otherwise.

See also

fgetpos(),

fseek(), and

ftell()

19.46 ftell()

This is a *File I/O* function. It is not implemented in the 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() and

fsetpos()

19.47 fwrite()

This is a *File I/O* function. It is not implemented in the 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(), and

fread()

19.48 getc()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
  #include <stdio.h >

  int getc(FILE *f);
```

Description

`getc()` is the same as `fgetc()`, but may be implemented as a macro. Therefore, make sure that `f` is not an expression having side effects. See `fgetc()` for more information.

19.49 getchar()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >


int getchar(void);
```

Description

`getchar()` is the same as getc() (`stdin`). See fgetc() for more information.

19.50 getenv()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >


char *getenv(const char *name);
```

Description

`getenv()` returns the value of environment variable `name`.

Return

`NULL`

19.51 gets()

This is a *File I/O* function. It is not implemented in the 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. This character is not appended to the string. The string is zero-terminated.

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 there was an error; `s` otherwise.

See also

fgetc() and

puts()

19.52 gmtime()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <time.h >


struct tm *gmtime(const time_t *time);
```

Description

`gmtime()` converts `*time` to UTC (Universal Coordinated Time), which is equivalent to GMT (Greenwich Mean Time).

Return

`NULL`, if UTC is not available; a pointer to a struct containing UTC otherwise.

See also

`ctime()` and

`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 following table describes the character ranges tested by the functions.

Table 19-6. Appropriate character range for the testing functions

Function	Range Tested
`isalnum()`	alphanumeric character, i.e., `A-Z`, `a-z` or `0-9`.
`isalpha()`	an alphabetic character, i.e., `A-Z` or `a-z`.
`iscntrl()`	a control character, i.e., `\000-\037` or `\177` (`DEL`).
`isdigit()`	a decimal digit, i.e., `0-9`.
`isgraph()`	a printable character except space (`! - or ~`).
`islower()`	a lower case letter, i.e., `a-z`.
`isprint()`	a printable character (`' '-'~'`).
`ispunct()`	a punctuation character, i.e., `'!'-'/'`, `':'-'@'`, `'['-'''` and `'{'-'~'`.
`isspace()`	a white space character, i.e., `' '`, `'\f'`, `'\n'`, `'\r'`, `'\t'` and `'\v'`.
`isupper()`	an upper case letter, i.e., `A-Z`.
`isxdigit()`	a hexadecimal digit, i.e., `0-9`, `A-F` or `a-f`.

Return

`TRUE` (i.e., 1), if `ch` is in the character class; zero otherwise.

See also

tolower() and

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.e., `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-2,147,483,648`, this value is returned and `errno` is set 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 it fails because the result would be too large, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

exp() and expf(),

frexp() and frexpf(),

log() and logf(),

log10() and log10f(), and

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 with the results of the division.

See also

div()

19.57 localeconv()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <locale.h >



struct lconv *localeconv(void);
```

Description

`localeconv()` returns a pointer to a `struct` containing information about the current locale, e.g., how to format monetary quantities.

Return

A pointer to a `struct` containing the desired information.

See also

setlocale()

19.58 localtime()

This is a *Hardware specific* function. It is not implemented in the 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(), and

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, `NAN` is returned; if it is equal to zero, `log()` returns negative infinity. In both cases, `errno` is set to `EDOM`.

See also

exp() and expf() and

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, `NAN` is returned; if it is equal to zero, `log10()` returns negative infinity. In both cases, `errno` is set to `EDOM`.

Seealso

exp() and expf() and

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. That location must have been marked by a call to setjmp(). The environment at the time of that call to `setjmp()` - `env`, which also was the parameter to setjmp() - is restored and your application continues as if the call to `setjmp()` just had returned the value `val`.

See also

`setjmp()`

19.62 malloc()

This is a *Hardware specific* function. It is not implemented in the 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 the default implementation in interrupt routines because it is not reentrant.

Return

`malloc()` returns a pointer to the allocated memory block. If the block could not be allocated, the return value is `NULL`.

See also

calloc() and

realloc()

19.63 mblen()

This is a *Hardware specific* function. It is not implemented in the 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() and

mbstowcs()

19.64 mbstowcs()

This is a *Hardware specific* function. It is not implemented in the 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 there was an error.

See also

mblen() and

mbtowc()

19.65 mbtowc()

This is a *Hardware specific* function. It is not implemented in the 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` there was an error.

See also

mblen(), and

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(), and

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(), and

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() and

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() and

`memcpy() and memmove()`

19.70 mktime()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
  #include <string.h >


  time_t mktime(struct tm *time);
```

Description

`mktime()` converts `*time` to a `time_t`. The fields of `*time` may have any value; they are not restricted to the ranges given `time.h`. If the conversion was 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(), and

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(), and

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.

See also

assert() and

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`, i.e., `xy`.

Return

`xy`, if `x > 0`

`1`, if `y == 0`

`+×`, if (`x == 0 && y < 0`)

`NAN`, if (`x < 0 && y` is not integral). Also, `errno` is set to `EDOM`.

`±×`, with the same sign as `x`, if the result is too large.

See also

exp() and expf(),

ldexp() and ldexpf(),

log() and logf(), and

modf() and modff()

19.74 printf()

This is a *File I/O* function. It is not implemented in the 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. If some error occurred, `EOF` is returned.

See also

fprintf() and

vfprintf(), vprintf(), and vsprintf()

19.75 putc()

This is a *File I/O* function. It is not implemented in the 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, make sure that the expression `f` has no unexpected effects. See fputc() for more information.

19.76 putchar()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >


int putchar(char ch);
```

Description

`putchar(ch)` is the same as putc `(ch, stdin)`. See fputc() for more information.

19.77 puts()

This is a *File I/O* function. It is not implemented in the 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 there was an error; zero otherwise.

See also

fputc() and

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 comparison function `cmp()` with two pointers to array elements. 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 19-7. Return value from the comparison function, cmp_func()

Key element value	Return value
less than the other one	less than zero (negative)
equal to the other one	zero
greater than the other one	greater than zero (positive)

The arguments to `qsort()` are listed in the following table:

Table 19-8. Possible arguments to the sorting function, qsort()

Argument Name	Meaning
`array`	A pointer to the beginning (i.e., 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 equal size.

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 not defined or a signal handler was not installed, the application is aborted.

Return

Non-zero, if there was an error; 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`.

19.81 realloc()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <stdlib.h >


void *realloc(void *ptr, size_t size);
```

Description

`realloc()` changes the size of a block of memory, 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 size of the memory block is smaller than the old size, `realloc()` discards that 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 the default implementation in interrupt routines because it is not reentrant.

Return

`realloc()` returns a pointer to the new memory block. If the operation cannot be performed, the return value is `NULL`.

See also

free()

19.82 remove()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdlib.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 there was an error; zero otherwise.

See also

tmpfile() and

tmpnam()

19.83 rename()

This is a *File I/O* function. It is not implemented in the 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 there was an error; zero otherwise.

See also

tmpfile() and

tmpnam()

19.84 rewind()

This is a *File I/O* function. It is not implemented in the 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.

See also

fopen(),

fseek(), and

fsetpos()

19.85 scanf()

This is a *File I/O* function. It is not implemented in the 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(), and

fscanf()

19.86 setbuf()

This is a *File I/O* function. It is not implemented in the 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; i.e., all input or output goes directly to and comes directly from the file. If `buf` is not `NULL`, it is used as a buffer (in that case, `buf` points to an array of `BUFSIZ` bytes).

See also

fflush() and

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 the 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 a longjmp().

See also

longjmp()

19.88 setlocale()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <locale.h>


char *setlocale(int class, const char *loc);
```

Description

`setlocale()` changes all or part of the program's locale, depending on `class`. The new locale is given by the character string `loc`. The classes allowed are given in the following table.

Table 19-9. Allowable classes for the setlocale() function

Class	Affected portion of program locale
`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

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(), and

strxfrm()

19.89 setvbuf()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >

void setvbuf(FILE *f,

             char *buf,

             int mode,

             size_t size);
```

Description

`setvbuf()` is used to specify how a file is buffered. `mode` determines how the file is buffered.

Table 19-10. Operating Modes for the setvbuf() Function

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() and

setbuf()

19.90 signal()

Syntax

```
#include <signal.h >
```

```
_sig_func signal(int sig, _sig_func handler);
```

Description

`signal()` defines how the application shall respond to the `sig` signal. The various responses are given in the following table.

Table 19-11. Various responses to the signal() function's input signal

Handler	Response to the signal
`SIG_IGN`	The signal is ignored.
`SIG_DFL`	The default response (`HALT`).
a function	The function is called with `sig` as parameter.

The signal handling function is defined as:

```
typedef void (*_sig_func)(int sig);
```

The signal can be raised 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` 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.

See also

raise()

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(), and

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(),

cosh() and coshf(),

sin() and sinf(), and

tan() and tanf()

19.93 sprintf()

Syntax

```
#include <stdio.h >
```

```
int sprintf(char *s, const char *format,...);
```

Description

`sprintf()` writes formatted output to the s string. It evaluates the arguments, converts them according to the specified 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 a 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>|#`

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 effect of "#" depends on the Conversion character, as listed in the following chapter.

Table 19-12. Effect of # in the Format specification

Conversion	Effect of "#"
`e, E, f`	The value of the argument always is printed with decimal point, even if there are no fractional digits.
`g, G`	As above, but In addition zeroes are appended to the fraction until the specified width is reached.
`o`	A zero is printed before the number to indicate an octal value.
`x, X`	"0x" (if the conversion is "x") or "0X" (if it is "X") is printed 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 " " 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 0s are used instead of blanks.

If an asterisk "*" is given, the field width is taken from the next argument, which of course 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, as listed in the following table.

Table 19-13. Effect of the Precision specification

Conversion	Precision
`d, i, o, u, x, X`	The minimum number of digits to print.

Table continues on the next page...

Table 19-13. Effect of the Precision specification (continued)

Conversion	Precision
`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`. Specifier `"L"` is used in conjunction with the conversion characters for floating point numbers and specifies `long double`.

  `Conversion = c|d|e|E|f|g|`

  `G|i|n|o|p|s|`

  `u|x|X|%`

The conversion characters have the following meanings, as the following table lists:

Table 19-14. Meaning of the Conversion Characters

Conversion	Description
`c`	The `int` argument is converted to `unsigned char`; the resulting character is printed.
`d, i`	An `int` argument is printed.
`e, E`	The argument must be a `double`. It is printed 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 is printed. The exponent always has at least 2 digits; the conversion character is printed just before the exponent.
`f`	The argument must be a `double`. It is printed in the form `[-]ddd.ddd` (see above). If the decimal point is printed, there is at least one digit to the left of it.
`g, G`	The 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. Scientific notation is used only if the exponent is < -4 or greater than or equal to the precision.

Table continues on the next page...

Table 19-14. Meaning of the Conversion Characters (continued)

Conversion	Description
`n`	The 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 `longint`.
`o`	The argument, which must be an `unsigned int`, is printed in octal notation.
`p`	The argument must be a pointer; its value is printed in hexadecimal notation.
`s`	The argument must be a `char *`; `sprintf()` writes the string.
`u`	The argument, which must be an `unsigned int`, is written in decimal notation.
`x, X`	The argument, which must be an `unsigned int`, is written in hexadecimal notation. `x` uses lower case letters `a` to `f`, while `X` uses upper case letters.
`%`	Prints a "`%`" sign. Give only as "`%%`".

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. The IEEE64 floating point implementation is not supported.

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.

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 meaning is given in the following table.

Table 19-15. 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 19-16. Description of the action taken for each conversion.

Conversion	Description
`c`	Reads a string of exactly `width` characters and stores it in the parameter. If no width is given, one character is read. The argument must be a `char *`. The string read is *not* zero-terminated.
`d`	A decimal number (syntax below) is read and stored in the parameter. The parameter must be a pointer to an integral type.
`i`	As `d`, 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 `long int`.
`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 assprintf() 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`	As `d`, but the parameter must be a pointer to an unsigned integral type.
`x`, `X`	As `u`, 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 (i.e., only characters *not* in the set are allowed). You can specify whole character ranges, e.g., `A-Z` specifies all upper-case letters. If you want 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. A `^` that shall be included in the list instead of indicating an inverted list must not be 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 uppercase character.`

- `[]abc]`

  `Allows ], a, b and c.`

- `[^]abc] Allows any char 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:

Listing: Syntax for Numbers

```
Number      = FloatNumber|IntNumber
IntNumber   = DecNumber|OctNumber|HexNumber

DecNumber   = Sign Digit {Digit}

OctNumber   = Sign 0 {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 (i.e., 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(), and

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.

Seealso

memchr(),

strrchr(), and

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()` is such that it could be used as a comparison function in bsearch() and qsort().

See also

memcmp(),

strcoll(), and

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(), and

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() and

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(), and

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'`), `s` is left unchanged and the function returns unsuccessfully. How the conversion is done is determined by the `format` string. This string contains text, which is copied one-to-one to `s`, and format specifiers. The latter always start with a `%` sign and are replaced by the following, as listed in the table below:

Table 19-17. strftime() output string content and format

Format	Replaced with
`%a`	Abbreviated name of the weekday of the current locale, e.g., `Fri`.
`%A`	Full name of the weekday of the current locale, e.g., `Friday`.

Table continues on the next page...

Table 19-17. strftime() output string content and format (continued)

Format	Replaced with
`%b`	Abbreviated name of the month of the current locale, e.g., `Feb`.
`%B`	Full name of the month of the current locale, e.g., `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.
`%y`	The year in short format, e.g., `"93"`.
`%Y`	The year, including the century (e.g., `"2007"`).
`%Z`	The time zone, if it can be determined.
`%%`	A single `'%'` sign.

Return

If the resulting string would have had more than `max` characters, zero is returned; otherwise the length of the created string is returned.

See also

mktime(),

setlocale(), and

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.

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() and

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, a `'\0'` is appended.

Return

`p`

See also

memcpy() and memmove() and

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(), and

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(), and

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(), and

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(), and

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`. The number format `strtod()` accepts is:

```
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, `0.0` is returned. If the value causes an overflow, `HUGE_VAL` is returned. In both cases, `errno` is set to `ERANGE`.

See also

atof(),

scanf(),

strtol(), and

strtoul()

19.114 strtok()

Syntax

```
#include <string.h >



char *strtok(char *p, const char *q);
```

Description

`strtok()` breaks the string `p` into tokens which are 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()` will continue at the position it stopped the previous time. `strtok()` saves the string `p` if it is not `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(), and

strstr()

19.115 strtol()

Syntax

```
#include <stdlib.h >

long strtol(const char *s, char **end, int base);
```

Description

`strtol()` converts string `s` into a `long int` 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`. The number format `strtol()` accepts is:

Listing: Number Format

```
Int_Number      = Dec_Number|Oct_Number|                    Hex_Number|Other_Num
Dec_Number      = SignDigit{Digit}
Oct_Number      = Sign0{OctDigit}
Hex_Number      = 0(x|X)Hex_Digit{Hex_Digit}
Other_Num       = 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`, base 8 (octal) is assumed. If it is `0x` or `0X`, base 16 (hexadecimal) is taken. Any other prefixes make `strtol()` scan a decimal number.

Return

The number read. If no number is found, zero is returned; if the value is smaller than `LONG_MIN` or larger than `LONG_MAX`, `LONG_MIN` or `LONG_MAX` is returned and `errno` is set to `ERANGE`.

See also

atoi(),

atol(),

scanf(),

strtod(), and

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`, 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`. The number format `strtoul()` accepts is the same as for strtol() except that the negative sign is not allowed, and so are the possible values for `base`.

Return

The number read. If no number is found, zero is returned; if the value is larger than `ULONG_MAX`, `ULONG_MAX` is returned and `errno` is set to `ERANGE`.

See also

atoi(),

atol(),

scanf(),

strtod(), and

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 would be longer than `n` characters, `p` is left unchanged.

Return

The length of the converted string.

See also

setlocale(),

strcmp(), and

strcoll()

19.118 system()

This is a *Hardware specific* function. It is not implemented in the Compiler.

Syntax

```
#include <string.h >

int system(const char *cmd);
```

Description

`system()` executes the `cmd` command line

Return

Zero

19.119 tan() and tanf()

Syntax

```
#include <math.h >

double tan(double x);

float  tanf(float x);
```

Description

`tan()` computes the tangent of `x`. Express `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(), and

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(), and

tan() and tanf()

19.121 time()

This is a *Hardware specific* function. It is not implemented in the 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(), and

strftime()

19.122 tmpfile()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
 #include <stdio.h >
FILE *tmpfile(void);
```

Description

`tmpfile()` creates a new temporary file using mode `"wb+"`. Temporary files automatically are deleted when they are closed or the application ends.

Return

A pointer to the file descriptor if the file could be created; `NULL` otherwise.

See also

fopen() and

tmpnam()

19.123 tmpnam()

This is a *File I/O* function. It is not implemented in the Compiler.

Syntax

```
#include <stdio.h >


char *tmpnam(char *s);
```

Description

`tmpnam()` creates a new unique filename. If `s` is not `NULL`, this name is assigned to it.

Return

A unique filename.

Seealso

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, the corresponding lower-case letter. Otherwise, `ch` is returned (unchanged).

Seealso

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit(),

toupper()

19.125 toupper()

Syntax

```
#include <ctype.h >
int toupper(int ch);
```

Description

`tolower()` 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, the corresponding upper-case letter. Otherwise, `ch` is returned (unchanged).

See also

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit(),

tolower()

19.126 ungetc()

This is a *File I/O* function. It is not implemented in the 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` will read that character.

Return

`ch`

See also

fgets(),

fopen(),

getc(), and

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 into an open parameter list. Calls to `va_arg()` get a parameter of the given type. The following listing shows how to do it:

Listing: Calling an open-parameter function

```
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. It is not implemented in the 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

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 in bytes of `s`.

Return

The length of `s` in bytes after the conversion.

See also

wcstombs()

19.130 wcstombs()

This is a *Hardware specific* function. It is not implemented in the 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 character in `s`, and returns the number of wide characters converted.

Return

The number of wide characters converted.

See also

wctomb()

Chapter 20
Appendices

The appendices covered in this manual are:

- 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 : Documentation for the entries in the mcutools.ini file
- Local Configuration File Entries : Documentation for the entries in the project.ini file.
- Known C++ Issues in RS08 Compilers : Documentation on known issues.
- RS08 Compiler Messages : Documentation on compiler messages.

Chapter 21
Porting Tips and FAQs

This appendix describes some FAQs and provides tips on the syntax of EBNF or how to port the application from a different tool vendor.

- Migration Hints
- General Optimization Hints
- Frequently Asked Questions (FAQs), Troubleshooting
- EBNF Notation
- Abbreviations, Lexical Conventions
- Number Formats
- Precedence and Associativity of Operators for ANSI-C
- List of all Escape Sequences

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 how to adapt them.

21.1.1 Porting from Cosmic

If your current application is written for Cosmic compilers, there are some special things to consider.

21.1.1.1 Getting Started

The best way is to create a new project using the New Project Wizard (in the CodeWarrior IDE: Menu *File* > *New*) or a project from a stationery template. This sets up a project for you with all the default options and library files included. Then add the existing files used for Cosmic to the project (e.g., through drag & drop from the Windows Explorer or using in the CodeWarrior IDE: the menu *Project* > *Add Files*. Make sure that the right memory model and CPU type are used as for 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). Enable this compiler option so the compiler accepts most Cosmic constructs.

21.1.1.3 Assembly Equates

For the Cosmic compiler, you need to define equates for the inline assembly using equ. If you want to use an equate or value in C as well, you need to define it using #define as well. For this compiler, you only need one version (i.e., use #define) both for C and for inline assembly. The equ directive is not supported in normal C code.

Listing: An example using the EQU directive

```
#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, for better type-safety with this compiler you need to place a `@' in front of variables if you want to use the address of a variable. Using a conditional block like the one below in the following listing may be difficult. Using macros which deal with the cases below is a better way to deal with this.

Listing: Using a conditional block to account for different compilers

```
#ifdef __MWERKS__
  ldx @myVariable,x
  jsr MyFunction
#else
  ldx _myVariable,x
  jsr _MyFunction
#endif
```

Listing: Using a macro to account for different compilers

```
#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` or another example (for the data section):

Listing: <codeph>#pragma DATA_SEG</codeph>

```
#ifdef __MWERKS__

#pragma DATA_SEG APPLDATA_SEG

#else

#pragma section {APPLDATA}

#endif
```

Listing: <codeph>#pragma CONST_SEG</codeph>

```
#ifdef __MWERKS__

#pragma CONST_SEG CONSTVECT_SEG

#else

#pragma section const {CONSTVECT}

#endif
```

Do not forget to 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:

Listing: Normal C constant syntax

```
#ifdef __MWERKS__

  and 0xF8

#else

  and #$F8

#endif
```

21.1.1.7 Inline Assembly and Index Calculation

Cosmic uses the + operator to calculate offsets into arrays. For the CodeWarrior IDE, you have to 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

Cosmic 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 need to remove the tab characters from such strings.

21.1.1.9 Inline Assembly and Operators

Cosmic's 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.

Listing: Accounting for different operators among 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

Cosmic uses the `@interrupt` syntax, whereas this compiler employs the `interrupt` syntax. In order to keep the source base portable, a macro can be used (e.g., 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) will handle the `#asm` blocks used in Cosmic inline assembly code Cosmic compatibility switch. However, if #asm is used with conditional blocks like `#ifdef` or `#if`, then the C parser may not accept it.

Listing: Use of Conditional Blocks without asm { and } Block Markers

```
void foo(void) {
  #asm
    nop
#if 1
  #endasm
  foo();
  #asm
    #endif
    nop
  #endasm
}
```

In such case, the `#asm` and `#endasm` must be ported to `asm {` and `}` block markers.

Listing: Use of Conditional Blocks with asm { and } Block Markers

```
void foo(void) {
  asm { // asm #1
    nop
#if 1
  } // end of asm #1
  foo();
  asm { // asm #2
#endif
    nop
  } // end of asm #2
}
```

21.1.1.12 Compiler Warnings

Check compiler warnings carefully. The Cosmic compiler does not warn about many cases where your application code may contain a bug. Later on the warnings can be switched off if desired (e.g., using the `-W2: Do Not Print Information or Warning Messages` option or using #pragma MESSAGE: Message Setting in the source code).

21.1.1.13 Linker *.prm File (for the Cosmic compiler) and Linker *.prm File (for this compiler)

Cosmic uses a *.prm file for the linker with a special syntax. 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 all your segments you declared in your application (through `#pragma DATA_SEG`, `#pragma CONST_SEG`, and `#pragma CODE_SEG`) are used in the `PLACEMENT` block of the linker prm file.

Check the linker warnings or errors carefully. They may indicate what you need to adjust or correct in your application. E.g., you may have allocated the vectors in the linker .prm file (using `VECTOR` or `ADDRESS` syntax) and allocated them as well in the application itself (e.g., with the `#pragma CONST_SEG` or with the `@address` syntax). Allocating objects twice is an error, so these objects must be allocated one or the other way, but not both.

Consult your map file produced by the linker to check that everything is correctly allocated.

Remember that the linker is a smart linker. This means that objects not used or referenced are not linked to the application. The Cosmic linker may link objects even if they are not used or referenced, but, nevertheless, these objects may still be required to be linked to the application for some reason not required by the linker. In order to have objects linked to the application regardless if they are used or not, 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

Allocation of bitfields is very compiler-dependent. Some compilers allocate the bits first from right (LSByte) to left (MSByte), and others allocate from left to right. Also, alignment and byte or word crossing of bitfields is not implemented consistently. Some possibilities are to:

- Check the different allocation strategies,
- Check if there is an option to change the allocation strategy in the compiler, or
- Use the compiler defines to hold sources portable:
    - `__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 that a particular compiler uses. Some compilers implement the sizes for the standard types (char, short, int, long, float, or double) differently. For instance, the size for an int is 16 bits for some compilers and 32 bits for others.

The sign of plain char is also not consistent for all compilers. If the software program requires that char be signed or unsigned, either change all plain char types to the signed or unsigned types or change the sign of char with the -T: Flexible Type Management option.

21.1.4 @bool Qualifier

Some compiler vendors provide a special keyword @bool to specify that a function returns a boolean value:

```
@bool int foo(void);
```

Because this special keyword is not supported, remove @bool or use a define such as this:

```
#define _BOOL /*@bool*/



_BOOL int foo(void);
```

21.1.5 @tiny and @far Qualifier for Variables

Some compiler vendors provide special keywords to place variables in absolute locations. Such absolute locations can be expressed in ANSI-C 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 will issue 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 will issue 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 ");`

This can be rewritten with `asm` or `asm {}`: `asm nop;`

21.1.9 Recursive Comments

Some compilers accept recursive comments without any warnings. The Compiler will issue a warning for each such recursive comment:

```
  /* this is a recursive comment /*

    int a;

  /* */
```

The Compiler will treat the above source completely 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. If there are such recursive comments, correct them.

21.1.10 Interrupt Function, @interrupt

Interrupt functions have to be marked with #pragma TRAP_PROC or using the interrupt keyword.

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 manual section discusses some important topics related to the handling of interrupt functions:

- Definition of an interrupt function
- Initialization of 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 pragma TRAP_PROC.
- Using the keyword interrupt.

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. In that case, the compiler terminates 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.

Listing: Example of using the "interrupt" keyword

```
interrupt void INCcount(void) {
  tcount++;

}
```

21.1.11.2 Initializing the Vector Table

Once the code for an interrupt function has been written, you must associated this function with an interrupt vector. This is done through initialization of the vector table. You can initialize the vector table in the following ways:

- Using the VECTOR ADDRESS or VECTOR command 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. You use the VECTOR ADDRESS command to write the address of a function at a specific address in the vector table.

In order 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
```

The VECTOR command is used to associate a function with a specific vector, identified with its number. The mapping from the vector number is target-specific.

In order to associate the address of the INCcount() function with the vector number 69, insert the following command in the application's PRM file.

Listing: Using the VECTOR command

```
VECTOR 69 INCcount
```

21.1.11.2.2 Using the interrupt Keyword

When you are using the keyword "interrupt", you may directly associate your interrupt function with a vector number in the ANSI C-source file. For that purpose, just specify the vector number next to the keyword interrupt.

In order to associate the address of the INCcount function with the vector number 75, define the function as in the following listing.

Listing: Definition of the INCcount() interrupt function

```
interrupt 75 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.

21.1.11.3.1 Defining a Function in a Specific Segment

In order to define a function in a specific segment, use the CODE_SEG pragma.

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 define where you want to allocate each segment you have defined in your source code. In order to place a segment in a specific memory area, just add the segment name in the PLACEMENT block of your PRM file. Be careful, as the linker is case-sensitive. Pay special attention to the upper and lower cases in your segment name.

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 General Optimization Hints

Here are some hints to reduce the size of your application:

- Find out if you need the full startup code. For example, if you do not have any initialized data, you can ignore or remove the copy-down. If you do not need any initialized memory, you can remove the zero-out. And if you do not need either, you may remove the complete startup code and set up your memory in the main routine. Use INIT main in the prm file as the startup or entry into your main routine of the application.
- Check the compiler options. For example, the -OdocF: Dynamic Option Configuration for Functions compiler option increases the compilation speed, but it decreases the code size. Using the -Li: List of Included Files to ".inc" File option to write a log file displays the statistics for each single option.
- Find out if you can use IEEE32 for both float and double. See the -T: Flexible Type Management option for how to configure this. Do not forget to link the corresponding ANSI-C library.
- Use smaller data types whenever possible (e.g., 8 bits instead of 16 or 32 bits).
- Look into the map file to check runtime routines, which usually have a `_' prefix. Check for 32-bit integral routines (e.g., _BMUL). Check if you need the long arithmetic.
- Enumerations: if you are using enums, by default they have the size of `int'. They can be set to an unsigned 8-bit (see option -T, or use -TE1uE).
- Check if you are using switch tables (have a look into the map file as well). There are options to configure this (see -CswMinSLB: Minimum Number of Labels for Switch Search Tables for an example).
- Finally, the linker has an option to overlap ROM areas (see the -COCC option in the linker).

21.3 Frequently Asked Questions (FAQs), Troubleshooting

This section provides some tips on how to solve the most commonly encountered problems.

21.3.1 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:

21.3.1.1 The Compiler Reports an Error, but WinEdit Does not Display it.

This means that WinEdit did not find the EDOUT file, i.e., the compiler wrote it to a place not expected by WinEdit. This can have several causes. Check that the DEFAULTDIR: Default Current Directory environment variable is not set and that the project directory is set correctly. Also in WinEdit 2.1, make sure that the OUTPUT entry in the file WINEDIT.INI is empty.

21.3.1.2 Some Programs Cannot Find a File.

Make sure the environment is set up correctly. Also check WinEdit's project directory. Read the Input Files section of the Files chapter.

21.3.1.3 The Compiler Seems to Generate Incorrect Code.

First, determine if the code is incorrect or not. Sometimes the operator-precedence rules of ANSI-C do not quite give the results one would expect. Sometimes faulty code can appear to be correct. Consider the example in the following listing:

Listing: Possibly faulty code?

```
if (x & y != 0) ...
evaluates as:
if (x & (y != 0)) ...
but not as:
if ((x & y) != 0) ...
```

Another source of unexpected behavior can be found among the integral promotion rules of C. Characters are usually (sign-)extended to integers. This can sometimes have quite unexpected effects, e.g., the if condition in the following listing is FALSE:

Listing: if condition is always FALSE

```
 unsigned char a, b;
  b = -8;

  a = ~b;

  if (a == ~b) ...
```

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.

21.3.1.4 The code seems to be correct, but the application does not work.

Check whether the hardware is not set up correctly (e.g., using chip selects). Some memory expansions are accessible only with a special access mode (e.g., only word accesses). If memory is accessible only in a certain way, use inline assembly or use the ` `volatile`' keyword.

21.3.1.5 The linker cannot handle an object file.

Make sure all object files have been compiled with the latest version of the compiler and with the same flags concerning memory models and floating point formats. If not, recompile them.

21.3.1.6 The make Utility does not Make the entire Application.

Most probably you did not specify that the target is to be made 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.

21.3.1.7 The make utility unnecessarily re-compiles a file.

This problem can appear if you have short source files in your application. It is caused by the fact that MS-DOS only saves the time of last modification of a file with an accuracy of ±2 seconds. If the compiler compiles two files in that time, both will 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 has to be recompiled. There is no way to solve this problem.

21.3.1.8 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. It is not compatible with the windows 3.1 version of WinHelp. The program `winhelp.exe` delivered with Windows 3.1, Windows 95 and Windows NT can only open Windows 3.1 help files. To open the compiler help file, use `Winhlp32.exe`.

The `winhlp32.exe` program resides either in the windows directory (usually `C:\windows`) 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 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* check box and select the `winhlp32.exe` file with the "other" button.

To change the association with the file manager under Windows 3.1 use the *File>Associate* menu entry.

21.3.1.9 How can constant objects be allocated in ROM?

Use #pragma INTO_ROM: Put Next Variable Definition into ROM and the -Cc: Allocate Const Objects into ROM compiler option.

21.3.1.10 The compiler cannot find my source file. What is wrong?

Check if in the default.env file the path to the source file is set in the environment variable GENPATH: #include "File" Path. In addition, you can use the -I: Include File Path compiler option to specify the include file path. With the CodeWarrior IDE, check the access path in the preference panel.

21.3.1.11 How can I switch off smart linking?

By adding a '+' after the object in the NAMES list of the prm file.

With the CodeWarrior IDE and the ELF/DWARF object-file format (see -F (-F2, -F2o): Object File Format) compiler option, you can link all in the object within an `ENTRIES...END` directive in the linker prm file:

```
ENTRIES fibo.o:* END
```

21.3.1.12 How to avoid the `no access to memory' warning?

In the simulator or debugger, change the memory configuration mode (menu *Simulator* > *Configure*) to `auto on access'.

21.3.1.13 How can the same memory configuration be loaded every time the simulator or debugger is started?

Save that memory configuration under default.mem. For example, select *Simulator* > *Configure* > *Save* and enter `default.mem`.

21.3.1.14 How can a loaded program in the simulator or debugger be started automatically and stop at a specified breakpoint?

Define the `postload.cmd` file. For example:

```
bs &main t


g
```

21.3.1.15 How can an overview of all the compiler options be produced?

Type in -H: Short Help on the command line of the compiler.

21.3.1.16 How can a custom startup function be called after reset?

In the prm file, use:

```
INIT myStartup
```

21.3.1.17 How can a custom name for the main() function be used?

In the prm file, use:

```
MAIN myMain
```

21.3.1.18 How can the reset vector be set to the beginning of the startup code?

Use this line in the prm file:

```
/* set reset vector on _Startup */


VECTOR ADDRESS 0xFFFE _Startup
```

21.3.1.19 How can the compiler be configured for the editor?

Open the compiler, select *File > Configuration* from the menubar, and choose *Editor Settings*.

21.3.1.20 Where are configuration settings saved?

In the `project.ini` file. With the CodeWarrior IDE, the compiler settings are stored in the `* .mcp` file.

21.3.1.21 What should be done when "error while adding default.env options" appears after starting the compiler?

Choose the options set by the compiler to those set in the default.env file and then save them in the project.ini file by clicking the save button in the compiler.

21.3.1.22 After starting up the ICD Debugger, an "Illegal breakpoint detected" error appears. What could be wrong?

The cable might be too long. The maximum length for unshielded cables is about 20 cm and it also depends on the electrical noise in the environment.

21.3.1.23 Why can no initialized data be written into the ROM area?

The const qualifier must be used, and the source must be compiled with the -Cc: Allocate Const Objects into ROM option.

21.3.1.24 Problems in the communication or losing communication.

The cable might be too long. The maximal length for unshielded cables is about 20 cm and it also depends on the electrical noise in the environment.

21.3.1.25 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.

21.3.1.26 How to get help on an error message?

Either press F1 after clicking on the message to start up the help file, or else copy the message number, open the pdf manual, and make a search on the copied message number.

21.3.1.27 How to 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 them. Or, otherwise, look into the manual for detailed information. A third way is to press F1 in the options setting dialog while a option is marked.

21.4 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. A short introduction to EBNF is presented.

Listing: EBNF Syntax

```
ProcDecl   = PROCEDURE "(" ArgList ")".

ArgList    = Expression {"," Expression}.

Expression = Term ("*"|"/") Term.

Term       = Factor AddOp Factor.

AddOp      = "+"|"-".

Factor     = (["-"] Number)|"(" Expression ")".
```

The EBNF language is a formalism that can be 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 how it is composed.

EBNF consists of the symbols discussed in the sections that follow.

- Terminal Symbols
- Non-Terminal Symbols
- Vertical Bar
- Brackets
- Parentheses
- Production End
- EBNF Syntax
- Extensions

21.4.1 Terminal Symbols

Terminal symbols (terminals for short) are the basic symbols which form the language described. In above example, the word `PROCEDURE` is a terminal. Punctuation symbols of the language described (not of EBNF itself) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.

21.4.2 Non-Terminal Symbols

Non-terminal symbols (non-terminals) are syntactic variables and have to be defined in a production, i.e., they have to appear on the left hand side of a production somewhere. In the example above, there are many non-terminals, e.g., `ArgList` or `AddOp`.

21.4.3 Vertical Bar

The vertical bar `"|"` denotes an alternative, i.e., either the left or the right side of the bar can appear in the language described, but one of them must appear. e.g., the 3rd production above means "an expression is a term followed by either a `"*"` or a `"/"` followed by another term."

21.4.4 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, i.e., 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 that the syntax in the example does not allow empty argument lists.)

21.4.5 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 (it is quoted) and may appear in the language.

21.4.6 Production End

A production is always terminated by a period.

21.4.7 EBNF Syntax

The definition of EBNF in the EBNF language is:

Listing: Definition of EBNF in the 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.

21.4.8 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. See the example in the following listing.

Listing: Example of a 4-byte identifier - FilePos

```
Struct = RefNo FilePos[4].
```

In some examples, text is enclosed by `"<"` and `">"`. This text is a meta-literal, i.e., whatever the text says may be inserted in place of the text (confer `<any char>` in the following listing, where any character can be inserted).

21.5 Abbreviations, Lexical Conventions

The following table has 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, e.g., float, double, long double
HLI	High-level Inline Assembly
Integral Type	Numerical type without a fractional part, e.g., char, short, int, long, long long

21.6 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 `e` or `E` has to be used. `-` and `+` can be used for signed representation of the floating number or the exponent.

The following suffixes are supported:

Table 21-2. Supported number suffixes

Constant	Suffix	Type
floating	`F`	`float`

Table continues on the next page...

Table 21-2. Supported number suffixes (continued)

Constant	Suffix	Type
floating	`L`	`long double`
integral	`U`	`unsigned in t`
integral	`uL`	`unsigned long`

Suffixes are not case-sensitive, e.g., `ul`, `Ul`, `uL` and `UL` all denote an `unsigned long` type. The following listing has 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 */
```

21.7 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

Table continues on the next page...

Table 21-3. ANSI-C Precedence and Associativity of Operators (continued)

Operators	Associativity
`|`	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.

The precedence and associativity is determined by the ANSI-C syntax (ANSI/ISO 9899-1990, p. 38 and Kernighan/ Ritchie, "*The C Programming Language*", Second Edition, Appendix Table 2-1).

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 above listing, operator-precedence causes the product of (`c*d`) to be added to `b`, and that sum is then assigned to `a`.

In the following listing, the associativity rules 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;
```

21.8 List of all Escape Sequences

The following table gives an overview over escape sequences which could be used inside strings (e.g., 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 documents the entries that can appear in the global configuration file. This file is named `mcutools.ini`.

`mcutools.ini` can contain these sections:

- [Options] Section
- [XXX_Compiler] Section
- [Editor] Section
- Example

22.1 [Options] Section

This section documents the entries that can appear in the `[Options]` section of the file `mcutools.ini`.

22.1.1 DefaultDir

Arguments

Default Directory to be used.

Description

Specifies the current directory for all tools on a global level (see also the DEFAULTDIR: Default Current Directory environment variable).

Example

```
DefaultDir=C:\install\project
```

22.2 [XXX_Compiler] Section

This section documents the entries that can appear in an `[`*XXX*`_Compiler]` section of the file `mcutools.ini`.

NOTE

XXX is a placeholder for the name of the actual backend. For example, for the RS08 compiler, the name of this section would be [RS08_Compiler].

22.2.1 SaveOnExit

Arguments

1/0

Description

Set to 1 to store configuration when the compiler is closed. Clear to 0 otherwise. 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 if not. The command line, its history, the windows 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 if not. The editor setting contains all information of the Editor Configuration dialog box.

22.2.4 SaveOptions

Arguments

1/0

Description

Set to 1 to save the options when writing a project file. Clear to 0 otherwise. The options also contain the message settings.

22.2.5 RecentProject0, RecentProject1, etc.

Arguments

Names of the last and prior project files

Description

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Example

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

22.2.6 TipFilePos

Arguments

Any integer, e.g., 236

Description

Actual position in tip of the day file. Used so different tips show at different calls.

Saved

Always saved when saving a configuration file.

22.2.7 ShowTipOfDay

Arguments

0/1

Description

Show the Tip of the Day dialog box at startup by setting ShowTipOfDay to 1.

1: Show Tip of the Day

0: Show only when opened in the help menu

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 entries that can appear in the `[Editor]` section of the `mcutools.ini` file.

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 box.

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 box, 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 box.

22.3.3 Editor_Opts

Arguments

The options to use 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 box.

Example

```
[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: A 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 2006 17:25:16
```

Chapter 23
Local Configuration File Entries

This appendix documents the entries that can appear in the local configuration file. Usually, you name this file *project*`.ini`, where *project* is a placeholder for the name of your project.

A *p*`roject.ini` file can contain these sections:

- [Editor] Section
- [XXX_Compiler] Section
- Example

23.1 Editor_Name

Arguments

The name of the local editor

Description

Specifies the name that is displayed for the local editor. This entry contains only a descriptive effect. Its content is not used to start the editor.

Saved

Only with Editor Configuration set in the **File > Configuration > Save Configuration** dialog box. This entry has the same format as the global Editor Configuration in the `mcutools.ini` file.

23.1.1 Editor_Name

Arguments

The name of the local editor

Description

Specifies the name that is displayed for the local editor. This entry contains only a descriptive effect. Its content is not used to start the editor.

Saved

Only with Editor Configuration set in the **File > Configuration > Save Configuration** dialog box. 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 box, the local editor selection is only active when this entry is present and not empty.

Saved

Only with Editor Configuration set in the **File > Configuration > Save Configuration** dialog box. This entry has the same format as for the global Editor Configuration in the `mcutools.ini` file.

23.1.3 Editor_Opts

Arguments

Local editor options

Description

Specifies options for the local editor to use. 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 box. 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 entries that can appear 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 RS08 compiler, the name of this section would be [RS08_Compiler].

23.2.1 RecentCommandLineX

NOTE

X is a placeholder for an integer.

Arguments

String with a command line history entry, e.g., `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 box.

23.2.2 CurrentCommandLine

Arguments

String with the command line, e.g., `fibo.c -w1`

Description

The currently visible command line content.

Saved

Only with Appearance set in the **File > Configuration >Save Configuration** dialog box.

23.2.3 StatusbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it afterwards.

Description

Is status bar currently enabled?

1: The status bar is visible

0: The status bar is hidden

Saved

Only with Appearance set in the **File > Configuration > Save Configuration** dialog box.

23.2.4 ToolbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it afterwards.

Description

Is the toolbar currently enabled?

1: The toolbar is visible

0: The toolbar is hidden

Saved

Only with Appearance set in the **File>Configuration > Save Configuration** dialog box.

23.2.5 WindowPos

Arguments

10 integers, e.g., " `0,1,-1,-1,-1,-1,390,107,1103,643`"

Special

This entry is only considered at startup. Later load operations do not use it afterwards.

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 box.

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 box.

Example

```
WindowFont=-16,500,0,Courier
```

23.2.7 Options

Arguments

-W2

Description

The currently active option string. This entry is quite long as the messages are also stored here.

Saved

Only with Options set in the **File > Configuration > Save Configuration** dialog box.

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

Saved

Only with Editor Configuration set in the **File > Configuration > Save Configuration** dialog box.

23.2.9 EditorCommandLine

Arguments

Command line for the editor.

Description

Command line content to open a file.

Saved

Only with Editor Configuration set in the **File > Configuration > Save Configuration** dialog box.

23.2.10 EditorDDEClientName

Arguments

Client command, e.g., `[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 box.

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 box.

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 box.

23.3 Example

The following listing shows a typical configuration file layout (usually *project*`.ini`):

Listing: A 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 RS08 Compilers

This appendix describes the known issues when using C++ with the RS08 compilers, and contains these sections:

24.1 Template Issues

This section describes unsupported template features.

- Template specialization is unsupported. Example:
  Listing: Example - Unsupported Template Specialization

```
template <class T> class C {};
template <> class C<double> {};

---------^-------------------- ERROR
```

- Declaring a template in a class is unsupported. Example:
  Listing: Example - Unsupported Declaration of Template in a Class

```
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:
  Listing: Example - Unsupported Non-template Parameters

```
    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:
  Listing: Example - Unsupported Implicit Instantiations

```
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:
  Listing: Example - Unsupported template Parameter

```
template <template <class P> class X, class T> struct A{}
```

- Defining a static function template is unsupported. Example:
  Listing: Example - Unsupported Static Function

```
   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:
  Listing: Example - Unsupported Operators in Expressions

```
- 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:
  Listing: Example - Using an Operator

```
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:
  Listing: Example - Unsupported Explicit Operator Calls

```
      struct B {
            operator int() { return 1; }

      };

      B b;

      b.operator int();

-------------^-------------ERROR: Not supported explicit operator call
```

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:
  Listing: Example - Unsupported Overloaded Operators

```
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:
  Listing: Example - Implemeting Unary ~ Operator

```
    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:
  Listing: Example - Implemeting Unary ! Operator

```
  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:
  Listing: Example - Unsupported OR Operator

```
class X {
public:

      operator int() {i = 1; return 1;}

} x;

(void) (0 || x);

-----------^------------ERROR
```

- Conditional operators are unsupported. Example:
  Listing: Example - Unsupported Conditional Operators

```
      int x = 1;
      int a = 2;

      int b = 3;

      x?a:b = 1;
```

```
-------------^------------------ERROR
```

- Assignment operators are incorrectly implemented. Example:
  Listing: Example - Incorrect Assignment Operator

```
      (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:
  Listing: Example - Defining a Pointer to Member Function Type

```
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.
  Listing: Example - Permitting an Implemetation

```
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:

Listing: Example - Unsupported %: Sequence Replacement

```
#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 RS08 compilers this type of class name definition causes an error. Example:

  Listing: Example - Unsupported Class Name Definition

```
class C { char c; };
void C(int x) { }

int x;

void main()

{

      C(x);

------^----------------------- ERROR

}
```

- Local classes are unsupported on the RS08 compilers. Example:

  Listing: Example - Unsupported Local Class

```
void f(void)
{

  class C {

    C() { }

  };

}
```

- The class member access feature is unsupported. Example:

  Listing: Example - Unsupported Class Member Access Feature

```
      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:
  Listing: Example - Unsupported Nested Class

```
      struct :: A a;
-------------------^----------------ERROR
```

- Function member definitions are not allowed within local class definitions. Example:
  Listing: Example - Unsupported Member Function Definition

```
void f (){
  class A{

        int g();

---------------^------Illegal local function definition

  };

}
```

- Defining a class within a function template is not allowed. Example:
  Listing: Example - Unsupported Definition of Class within a Function Template

```
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:

  Listing: Example - Unsupported Scope

```
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:

  Listing: Example - Unimplemented Storage Class Specifiers

```
        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 RS08 compilers.

- Pointer to Member
  - Global pointer to member initialization is unimplemented. Example:
    Listing: Example - Unsupported Pointer to Member

```
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:

  Listing: Example - Unsupported Initialization

```
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:

  Listing: Example - Unsupported Arrary Construction

```
int S::* a0[3];
a0[1] = &S::i

---------^-----------------Failed
```

- Static member - When you refer to a static member using the class member access syntax, the object-expression is not evaluated or is evaluated incorrectly. Example:

  Listing: Example - Static Member

```
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:

    Listing: Example - Unsupported Non-Static Member Functions

```
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 RS08 compiler, it does not. Example:

    Listing: Example - Unsupported Non-Static Data Member

```
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:

    Listing: Example - Accessing a protected Member Using a Friend Function

```
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:
  Listing: Example - Specifying a Private Nested Type as the Return Type

```
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:
  Listing: Example - Accessing a Protected Member

```
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:

  Listing: Example - Access Modification of a Base Class Member

```
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:
  Listing: Example - Unsupported Copy Constructor

```
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:
  Listing: Example - Unsupported Usage of Non-Expilcit Constructor

```
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:
  Listing: Example - Unsupported Direct Invoking of a Virtual Function

```
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:
  Listing: Example - Unsupported Indirect Invoking of a Virtual Function

```
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:
  Listing: Unsupported Invoking of a Virtual Function Defined in Derived Class

```
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:

  Listing: Example - Unsupported Overloaded Declaration

```
      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:

  Listing: Example - Unsupported Address of Overloaded Function

```
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:
  Listing: Example - Unsupported Pointer to Member Function Return Value

```
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:
  Listing: Example - Unsupported Overloaded Name

```
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:
  Listing: Example - Unsupported Operator Overloading

```
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:
  Listing: Example - Unsupported Implicit Conversions

  ```
  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 RS08 compilers. Example:
  Listing: Example - Unsupported Initialization

  ```
  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
  ```

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:

  Listing: Example - Standard Conversion Sequence

```
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:

  Listing: Example - Standard Conversion Sequence2

```
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:

  Listing: Example - Pointer Conversion with a Conversion Rank

```
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:

  Listing: Example - User-Defined Conversion Sequence

```
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:
  Listing: Example - Implicit conversion sequence

```
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:
  Listing: Example 2 - Implicit conversion sequence

```
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 sub-sequence of the other, and when `s1` has Promotion rank, and `s2` has Conversion rank, then overload resolution prefers `s1` to `s2`. Example:
  Listing: Example 3 - Implicit conversion sequence

```
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 RS08 compiler:

Listing: Example - Explicit Type Conversions

```
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:
  Listing: Example 2 - Explicit Type Conversions

  ```
  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:
  Listing: Example 3 - Explicit Type Conversions

  ```
  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:
  Listing: Example 4 - Explicit Type Conversions

```
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:
  Listing: Example - Unsupported Initialization Features

```
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:

Listing: Example 2 - Unsupported Initialization Features

```
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:

  Listing: Example 3 - Unsupported Initialization Features

```
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:

  Listing: Example 4 - Unsupported Initialization Features

```
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 RS08 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:
  Listing: Example - Unsupported Scope

  ```
  switch (a){
        case 'a': break;

        default :

              int x = 1;

        ----------^--------------ERROR: Not declared x

  }
  ```

- An `if` condition with initialized declaration is unsupported. Example:
  Listing: Example - Unsupported Initialization Features

  ```
  if(int i = 0)
  ------^-----------------ERROR
  ```

The following internal errors occur when using C++ with the RS08 compiler:

- Internal Error #103. Example:
  Listing: Example - Internal Error #103

  ```
  long double & f(int i ) {return 1;}
  long double i;

  if (f(i)!=i)

  --------^-----------------Internal Error
  ```

- Internal Error #385, generated by the following example:
  Listing: Example - Internal Error #385

```
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:

  Listing: Example - Internal Error #418

```
#include <time.h>
struct std::tm T;
```

- Internal Error #604, generated by the following example:

  Listing: Example - Internal Error #604

```
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:

  Listing: Example - Internal Error #1810

```
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:
    Listing: Example - Namespaces

```
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:
    Listing: Example - Name Lookup Feature

```
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:
  Listing: Example - Enumerator Declaration

  ```
  enum {one=1, two, hidden_name };
  struct hidden_name{int x;};

  -----------^----------------Not allowed
  ```

- Global initializers with non-`const` variables are unsupported. Example:
  Listing: Example - Global Initializers

```
int x;
int y = x;
```

- Anonymous unions are unsupported. Example:
  Listing: Example - Anonymous Unions

```
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:
  Listing: Example - Fundamental Type Feature

```
      int fun (char x){}
      int fun (unsigned char x){}

--------------^--------------------------Illegal function redefinition
```

- Enumeration declaration features
  - Defining an enum in a local scope of the same name is unsupported. Example:

Listing: Example - Unsupported Enumeration Declaration

```
          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:
  Listing: Example 2 - Unsupported Enumeration Declaration

```
int fun(short l) { return 0; }
int fun(const int l) { return 1; }

enum E { x, y };

fun(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:
    Listing: Example - Unsupported Union

```
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:
    Listing: Example - Multiple Base Definition

```
Class B:public A(){};
Class C: public B(){};

Class D :public B, public A,publicC{};
```

  - Multiple virtual base classes. Example:
    Listing: Example - Multiple Virtual Base

```
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:
  Listing: Example - Unsupported Friend Function

  ```
  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:
  Listing: Example - Call to Named Function

  ```
  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:
  Listing: Example - Preprocessing Directives

  ```
  #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:
    Listing: Example - Line Control

```
#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:

    Listing: Example - Viable Function

    ```
    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:

    Listing: Example - Reference

    ```
              const X& r = 4;
    -----------------------^------------ERROR: Illegal cast-operation
    ```
  - The following syntax is unsupported:

    Listing: Example - Unsupported Syntax

    ```
       int a7, a;
       if(&(::a7) == &a);
    ```

```
---------^-------------------ERROR:Not supported operator ::
```

- Aggregate features
  - Object initialization fails. Example:

    Listing: Example - Unsupported Object Initialization

    ```
    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:

    Listing: Example 2 - Unsupported Object Initialization

    ```
    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:

  Listing: Example - Unsupported typedef Specifier

  ```
  typedef int  new_type;
  typedef int  new_type;
  ```

```
--------------^-------ERROR: Invalid redeclaration of new_type
```

- This return statement causes an error:
  Listing: Example - Unsupported Return Statement

```
return ((void) 1);
----------------------------^------------ERROR
```

- Permitting a function to appear in an integral constant if it appears in a `sizeof` expression is unsupported. Example:
  Listing: Example - Unsupported Expression

```
void f() {}
int i[sizeof &f];

--------------------^-------------ERROR
```

- Defining a local scope using a compound statement is an unimplemented feature. Example:
  Listing: Example - Local Scope

```
int i = 4;
int main(){

      if ((i != 1) || (::i != 4));

------------------------^----------ERROR

}
```

- The following Main function is currently unimplemented:
  Listing: Example - Unimplemented Main Function

```
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:
    Listing: Example - Unsupported Function Call

```
int main(){
      int f(void);

int (&fr)(void) = f;/

}
```

  - Return pointer type of a function make ambiguous between `void *` and `X *`. Example:

Listing: Example 2 - Unsupported Function Call

```
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:

    Listing: Example 3 - Unsupported Function Call

```
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:

    Listing: Example - Unsupported Enumeration

```
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:

    Listing: Example - Restrictions for Delete Operation

```
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:

  Listing: Example 2 - Restrictions for Delete Operation

  ```
    S * ps1 = new S[5];
    ::delete [] ps1;

  ------------^------ERROR: ~S is not used
  ```

- Error at declaring delete operator. Example:

  Listing: Example 3 - Restrictions for Delete Operation

  ```
     void operator delete[](void *p){};
  -----------------------^--------------ERROR
  ```

- The New operator is unimplemented. Example:

  Listing: Example - Unimplemented New Operator

```
- void * operator new[](size_t);
  -------------------^--------ERROR: Operator must be a function
```

- The following Expression fails to initialize the object. Example:

  Listing: Example - Failed Initialization of the Object

```
int *p = new int(1+(2*4)-3);
-----------------^-----ERROR: The object is not initialized
```

- Use placement syntax for new `int` objects. Example:

  Listing: Example - Using Placement Syntax for New int Objects

```
int * p1, *p2;
p1 = new int;

p2 = new (p1) int;

--------------^----------------ERROR: Too many arguments
```

- The following Multi-dimensional array syntax is not supported:
  Listing: Example 7 - Unsupported Multi-dimensional Array Syntax

```
      int  tab[2][3];
      int fun(int (*tab)[3]);

-------------------^---------------------ERROR
```

- The following Goto syntax is unsupported:
  Listing: Example - Unsupported Goto Syntax

```
            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:
    Listing: Example - Unsupported Function Syntax

```
    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:
    Listing: Example 2 - Unsupported Function Syntax

```
    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:
    Listing: Example 3 - Unsupported Function Syntax

```
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:
  Listing: Example 4 - Unsupported Function Syntax

```
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:
  Listing: Example - Unsupported Character Literals

```
     int f(int i, char c) {return 1;}
     f('abcd', 'c');

-----------^--------------------ERROR
```

- The String characteristic "A string is an `array of `nconstchar`'" is not supported. Example:
  Listing: Example - Unsupported String Characterstic

```
int type(const char a[]){return 1};
type("five") != 1 /*Runtime failed*/
```

- Ambiguity Resolution
  Listing: Example - 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:

  Listing: Example - Unsupported const Reference

```
             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:

  Listing: Example - Invalid Jump

```
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
RS08 Compiler Messages

This chapter describes the compiler messages.

NOTE

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

25.1 Compiler Messages

Following is the list of the compiler messages.

25.1.1 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.

25.1.2 C2: Message overflow, skipping <kind> messages

[INFORMATION]

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.

25.1.3 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).

25.1.4 C51: Cannot open statistic log file <file>

[WARNING]

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.

25.1.5 C52: Error in command line <cmd>

[FATAL]

Description

In case there is an error while processing the command line, this message is issued.

25.1.6 C53: Message <Id> is not used by this version. The mapping of this message is ignored.

[WARNING]

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?

25.1.7 C54: Option <Option> .

[INFORMATION]

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.

25.1.8 C56: Option value overriden for option <OptionName>. Old value `<OldValue>', new value `<NewValue>'.

[WARNING]

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'.*/
```

25.1.9 C64: Line Continuation occurred in <FileName>

[INFORMATION]

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>".

25.1.10 C65: Environment macro expansion message '<description>' for <variablename>

[INFORMATION]

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.

25.1.11 C66: Search path <Name> does not exist

[INFORMATION]

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.

25.1.12 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.

25.1.13 C1001: Multiple const declaration makes no sense

[WARNING]

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

25.1.14 C1002: Multiple volatile declaration makes no sense

[WARNING]

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.

25.1.15 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.

25.1.16 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.

25.1.17 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

25.1.18 C1006: Illegal storage class

[WARNING]

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.

25.1.19 C1007: Type specifier mismatch

[ERROR]

Description

The type declaration is wrong.

Example

```
  int float i;
```

Tips

Do not use an illegal type chain.

25.1.20 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.

25.1.21 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.

25.1.22 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.

25.1.23 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
}
```

25.1.24 C1013: Old style declaration

[WARNING]

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) {

  ...

}
```

25.1.25 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.

25.1.26 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.

25.1.27 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.

25.1.28 C1017: Empty declaration

[ERROR]

Description

A declaration cannot be empty.

Example

```
int;
```

Tips

There must be a name for an object.

25.1.29 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.

25.1.30 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.

25.1.31 C1020: Incompatible type to previous declaration

[WARNING]

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.

25.1.32 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.

Seealso

- C1106: Non-standard bitfield type

25.1.33 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.

25.1.34 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.

25.1.35 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).

25.1.36 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 '...'.

25.1.37 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.

25.1.38 C1027: Reference must be initialized

[ERROR]

Description

A reference was not initialized when it was declared.

Example

```
  int j;

  int& i // = j; missing
```

Tips

Initialize the reference with an object of the same type as the reference points to.

25.1.39 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.

25.1.40 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.

25.1.41 C1030: Pointer to reference illegal

[ERROR]

Description

A pointer to a reference was declared.

Example

```
void f(int & * p);
```

Tips

The variable must be dereferenced before a pointer to it can be declared.

25.1.42 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.

25.1.43 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).

25.1.44 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.

25.1.45 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.

25.1.46 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.

25.1.47 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.

25.1.48 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.

25.1.49 C1038: Cannot be friend of myself

[WARNING]

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.

25.1.50 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.

25.1.51 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.

25.1.52 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).

25.1.53 C1042: Multiple virtual declaration makes no sense

[WARNING]

Description

The specified class or structure was declared as virtual more than once.

Tips

Use only one virtual modifier for each base class.

25.1.54 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.

25.1.55 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.

25.1.56 C1045: <Special member function> not generated

[WARNING]

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.

25.1.57 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.

25.1.58 C1047: Local compiler generated <Special member function> not supported

[WARNING]

Description

Local class declarations would force the compiler to generate local special member functions (Copy Constructor, Default Constructor, Destructor, Assignment \c 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.

25.1.59 C1048: Generate compiler defined <Special member function>

[INFORMATION]

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&);

    A& operator =(const A& );
    ~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.

25.1.60 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.

25.1.61 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.

25.1.62 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 "}".

25.1.63 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.

25.1.64 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.

25.1.65 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.

25.1.66 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.

25.1.67 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.

25.1.68 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 *

25.1.69 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 *

25.1.70 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

25.1.71 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

25.1.72 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.

25.1.73 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.

25.1.74 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.

25.1.75 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.

25.1.76 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.

25.1.77 C1066: Destructor must not have return type

[ERROR]

Description

The specified destructor returned a value.

Tips

Do not declare a return type for destructors.

25.1.78 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++.

25.1.79 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.

25.1.80 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.

25.1.81 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.

25.1.82 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.

25.1.83 C1072: Redefined extern to static

[WARNING]

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.

25.1.84 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.

25.1.85 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.

25.1.86 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.

25.1.87 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.

25.1.88 C1077: <FunctionKind> must not have parameters

[ERROR]

Description

Parameters were declared where it is illegal.

Tips

Do not declare parameters for Destructors and Conversions.

25.1.89 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.

25.1.90 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.

25.1.91 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.

25.1.92 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.

25.1.93 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.

25.1.94 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.

25.1.95 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.

25.1.96 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.

25.1.97 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.

25.1.98 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.

25.1.99 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.

25.1.100 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.

25.1.101 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.

25.1.102 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.

25.1.103 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.

25.1.104 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.

25.1.105 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).

25.1.106 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.

25.1.107 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.

25.1.108 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.

25.1.109 C1100: Reference to void illegal

[ERROR]

Description

The specified identifier was declared as a reference to void.

Example

```
void &amp;vr;  // error;
```

Tips

Do not declare references to a void type.

25.1.110 C1101: Reference to bitfield illegal

[ERROR]

Description

A reference to the specified bit field was declared.

Example

```
struct A {

  int &i : 3; // error

};
```

Tips

Do not declare references to bitfields.

25.1.111 C1102: Array of reference illegal

[ERROR]

Description

A reference to the specified array was declared.

Example

```
extern int &j[20];
```

Tips

Do not declare references to arrays.

25.1.112 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.

25.1.113 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.

25.1.114 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.

25.1.115 C1106: Non-standard bitfield type

[WARNING]

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.

25.1.116 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.

25.1.117 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.

25.1.118 C1109: Generate call to Copy Constructor

[INFORMATION]

Description

An instance of a class was passed as argument of a function, needing to be copied onto the stack with 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 &);

  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.

25.1.119 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.

25.1.120 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.

25.1.121 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

25.1.122 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

25.1.123 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.

25.1.124 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

25.1.125 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.

25.1.126 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.

25.1.127 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.

25.1.128 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.

25.1.129 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.

25.1.130 C1121: Definition needed if called with explicit scope resolution

[INFORMATION]

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.

25.1.131 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;

}
```

25.1.132 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);
}
```

25.1.133 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);

}
```

25.1.134 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;

}
```

25.1.135 C1126: Abstract class cause inheriting pure virtual without overriding function(s)

[INFORMATION]

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

};
```

25.1.136 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.

25.1.137 C1128: Class contains private members only

[WARNING]

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!

25.1.138 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);
```

25.1.139 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

25.1.140 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();

}
```

25.1.141 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.

25.1.142 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
```

25.1.143 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

    };

  }
```

25.1.144 C1135: Ignore Storage Class Specifier cause it only applies on objects

[WARNING]

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;
```

25.1.145 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{};
```

25.1.146 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

25.1.147 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

25.1.148 C1139: Interrupt routine should not have any return value nor any parameter

[WARNING]

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){ }
```

25.1.149 C1140: This function is already declared and has a different prototype

[WARNING]

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).

25.1.150 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.

25.1.151 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.

25.1.152 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.

25.1.153 C1144: Multiple restrict declaration makes no sense

[WARNING]

Description

The restrict qualifier should only be applied once and not several times.

Example

```
  int * restrict restrict pointer;
```

Tips

Only specify restrict once.

25.1.154 C1390: Implicit virtual function

[INFORMATION]

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.

25.1.155 C1391: Pseudo Base Class is added to this class

[INFORMATION]

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).

25.1.156 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

25.1.157 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

25.1.158 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;

  }
```

25.1.159 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;
```

```
}
```

25.1.160 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;

  }
```

25.1.161 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.

Seealso

- option -T
- C++ Front End.

25.1.162 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.

25.1.163 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.

25.1.164 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.

25.1.165 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.

25.1.166 C1404: Return expected

[WARNING]

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.

25.1.167 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.

25.1.168 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.

25.1.169 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.

25.1.170 C1408: Incorrect function-definition

[ERROR]

Description

The function definition is not correct or ill formed.

Tips

Correct the source.

25.1.171 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.

25.1.172 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.

25.1.173 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.

25.1.174 C1412: Not a function call, address of a function

[WARNING]

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.

25.1.175 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.

25.1.176 C1414: Casting to pointer of non base class

[WARNING]

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.

25.1.177 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.

25.1.178 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.

25.1.179 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.

25.1.180 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 [.

25.1.181 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.

25.1.182 C1420: Result of function-call is ignored

[WARNING]

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

}
```

25.1.183 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.

25.1.184 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
```

25.1.185 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.

25.1.186 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.

25.1.187 C1425: No Destructor available to call

[WARNING]

Description

No destructor is available, but one must be called.

25.1.188 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.

25.1.189 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.

25.1.190 C1428: No wide characters supported

[WARNING]

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.

25.1.191 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.

25.1.192 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.

25.1.193 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.

25.1.194 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.

25.1.195 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.

25.1.196 C1434: This C++ feature is not yet implemented

[WARNING]

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.

25.1.197 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.

25.1.198 C1436: delete needs number of elements of array

[WARNING]

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[].

25.1.199 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;

    ...

  }
```

25.1.200 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)();

}
```

25.1.201 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.

See also

- if __OPTION_ACTIVE__

25.1.202 C1440: This is causing previous message <MsgNumber>

[INFORMATION]

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.

25.1.203 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).

25.1.204 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.

25.1.205 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").

25.1.206 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;

      }
```

25.1.207 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;

    }
```

25.1.208 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.

25.1.209 C1801: Implicit parameter-declaration for '<FuncName>'

[WARNING]

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();

}
```

25.1.210 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.

25.1.211 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.

25.1.212 C1804: Ident expected

[ERROR]

Description

An identifier was expected.

Example

```
int ;
```

25.1.213 C1805: Non standard conversion used

[WARNING]

Description

An object pointer is cast to a function pointer or vice-versa - which is not allowed in ANSI-C. 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

}
```

25.1.214 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.

25.1.215 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.

25.1.216 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

25.1.217 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.

25.1.218 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.

25.1.219 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.

25.1.220 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.

25.1.221 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))
```

25.1.222 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.

25.1.223 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.

25.1.224 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.

25.1.225 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.

25.1.226 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.

25.1.227 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!

25.1.228 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.

25.1.229 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.

25.1.230 C1823: Undefining an implicit parameter-declaration

[WARNING]

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.

25.1.231 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.

25.1.232 C1825: Indirection to different types

[WARNING]

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.

25.1.233 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.

25.1.234 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.

25.1.235 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.

25.1.236 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!

25.1.237 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.

25.1.238 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!.

25.1.239 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)++;

}
```

25.1.240 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.

25.1.241 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.

25.1.242 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.

25.1.243 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.

25.1.244 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.

25.1.245 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.

25.1.246 C1839: Variable of type struct or union expected

[ERROR]

Description

A variable of structure or union type was expected.

25.1.247 C1840: Pointer to struct or union expected

[ERROR]

Description

A pointer to a structure or union was expected.

25.1.248 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.

25.1.249 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.

25.1.250 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.

25.1.251 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.

25.1.252 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.

25.1.253 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.

25.1.254 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.

25.1.255 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.

25.1.256 C1850: Incompatible pointer operands

[ERROR]

Description

Pointer operands were incompatible.

Tips

Either change the source or explicitly cast the pointers.

25.1.257 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.

25.1.258 C1852: Illegal sizeof operand

[ERROR]

Description

The sizeof operand was a bitfield.

Tips

Do not use bitfields as sizeof operands.

25.1.259 C1853: Unary minus operator applied to unsigned type

[WARNING]

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.

25.1.260 C1854: Returning address of local variable

[WARNING]

Description

An address of a local variable is returned.

Example

```
int &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)!

25.1.261 C1855: Recursive function call

[WARNING]

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.

25.1.262 C1856: Return <expression> expected

[WARNING]

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 {

      ...

    }

    ...

  }
```

25.1.263 C1857: Access out of range

[WARNING]

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 = &buf[3]; // no message!
  buf[4] = 0;  // message
```

25.1.264 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.

25.1.265 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;

}
```

25.1.266 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.

25.1.267 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'.

25.1.268 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.

25.1.269 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.

25.1.270 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.

25.1.271 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.

25.1.272 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.

25.1.273 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.

25.1.274 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.

25.1.275 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.

25.1.276 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.

25.1.277 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

25.1.278 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

25.1.279 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

25.1.280 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

25.1.281 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.

25.1.282 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.

25.1.283 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.

25.1.284 C2018: Generate class/struct from template

[INFORMATION]

Description

A class was instantiated from a template.

Example

```
template<class S> struct A {

  S s;

};

A<int> a;  // information
```

25.1.285 C2019: Generate function from template

[INFORMATION]

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.

25.1.286 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.

25.1.287 C2021: Generate NULL-check for class pointer

[INFORMATION]

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.

25.1.288 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

    //...

  }
```

25.1.289 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.

25.1.290 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.

25.1.291 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.

25.1.292 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

25.1.293 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

25.1.294 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

25.1.295 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

25.1.296 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

25.1.297 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

25.1.298 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

25.1.299 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

25.1.300 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.

See also

- Limitations

25.1.301 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

25.1.302 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

25.1.303 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

25.1.304 C2401: Pragma <ident> expected

[WARNING]

Description

A single pragma was found.

Example

```
#pragma
```

Tips

Probably this is a bug. Correct it.

25.1.305 C2402: Variable <Ident> <State>

[DISABLE]

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.

25.1.306 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

25.1.307 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

25.1.308 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

25.1.309 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"

25.1.310 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.

Seecalso

- Number Formats
- header file "float.h"

25.1.311 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

25.1.312 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

25.1.313 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"

25.1.314 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.

25.1.315 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.

25.1.316 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.

25.1.317 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.

25.1.318 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

25.1.319 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.

25.1.320 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.

25.1.321 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

25.1.322 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.

Seealso

- Loop Unrolling

25.1.323 C2901: Unrolling loop

[INFORMATION]

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 -Cu or pragma LOOP_UNROLL) 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

25.1.324 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

25.1.325 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.

25.1.326 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

25.1.327 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.

25.1.328 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!

25.1.329 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

25.1.330 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

25.1.331 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

25.1.332 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 allo cate 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.

25.1.333 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

25.1.334 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

25.1.335 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";
```

25.1.336 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.

25.1.337 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.

25.1.338 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.

25.1.339 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

25.1.340 C3602: Global objects: <Number>, Data Size (RAM): <Size>, Const Data Size (ROM): <Size>

[INFORMATION]

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.

25.1.341 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.

25.1.342 C3604: Static '<Object>' was not referenced

[INFORMATION]

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.

25.1.343 C3605: Runtime object '<Object>' is used at PC <PC>

[DISABLE]

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.

25.1.344 C3606: Initializing object '<Object>'

[DISABLE]

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.

25.1.345 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.

25.1.346 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.

25.1.347 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.

25.1.348 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.

25.1.349 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
```

25.1.350 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.

25.1.351 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.

25.1.352 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

25.1.353 C4000: Condition always is TRUE

[INFORMATION]

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();

  }

}
```

25.1.354 C4001: Condition always is FALSE

[INFORMATION]

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

25.1.355 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.

25.1.356 C4003: Shift count converted to unsigned char

[INFORMATION]

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.

25.1.357 C4004: BitSet/BitClr bit number converted to unsigned char

[INFORMATION]

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.

25.1.358 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

25.1.359 C4007: Pointer DREF is NOT allowed

[INFORMATION]

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).

25.1.360 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.

25.1.361 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.

25.1.362 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.

25.1.363 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.

25.1.364 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

25.1.365 C4203: Invalid pragma MESSAGE, <description>

[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

25.1.366 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

25.1.367 C4205: Invalid pragma LINK_INFO, <description>

[WARNING]

Description

The pragma LINK_INFO was used in a ill formed way.

See also

- pragma LINK_INFO

25.1.368 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

25.1.369 C4207: Invalid pragma pop, <description>

[WARNING]

Description

The pragma pop was used in a ill formed way.

See also

- pragma pop
- pragma push

25.1.370 C4208: Invalid pragma push, <description>

[WARNING]

Description

The pragma push was used in a ill formed way.

Seealso

- pragma pop
- pragma push

25.1.371 C4209: Usage: pragma align (on|off)

[WARNING]

Description

The pragma align was not used together with on or off.

See also

- pragma align

25.1.372 C4300: Call of an empty function removed

[WARNING]

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

25.1.373 C4301: Inline expansion done for function call

[INFORMATION]

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

25.1.374 C4302: Could not generate inline expansion for this function call

[INFORMATION]

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.

25.1.375 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.

25.1.376 C4400: Comment not closed

[FATAL]

Description

The preprocessor has found a comment which has not been closed.

Example

```
/*
```

Tips

Close the comment.

25.1.377 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 */
```

25.1.378 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).

25.1.379 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

25.1.380 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 ')'.

25.1.381 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.

25.1.382 C4406: Closing '>' missing

[FATAL]

Description

There is a missing closing '>' for the include directive.

Example

```
#include <string.h
```

Tips

Correct the include directive.

25.1.383 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.

25.1.384 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

25.1.385 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.

25.1.386 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.

25.1.387 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

25.1.388 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

25.1.389 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).

25.1.390 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.

25.1.391 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.

25.1.392 C4416: Comma expected

[FATAL]

Description

The preprocessor expects a comma at the given position.

Tips

Check your macro definition or usage.

25.1.393 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.

25.1.394 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

25.1.395 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.

25.1.396 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.

25.1.397 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

25.1.398 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.

25.1.399 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

25.1.400 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.

25.1.401 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.

25.1.402 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.

25.1.403 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.

25.1.404 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

25.1.405 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.

25.1.406 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.

25.1.407 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.

25.1.408 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.

25.1.409 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).

25.1.410 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.

25.1.411 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.

25.1.412 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.

25.1.413 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.

25.1.414 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.

25.1.415 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.

25.1.416 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.

25.1.417 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.

25.1.418 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.

25.1.419 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
```

25.1.420 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.

25.1.421 C4445: Line numberfor 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.

25.1.422 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.

25.1.423 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.

25.1.424 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.

25.1.425 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?

25.1.426 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.

25.1.427 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.

25.1.428 C4800: Implicit cast in assignment

[WARNING]

Description

An assignment requiring an implicit cast was made.

Tips

Check, if the casting results in correct behavior.

25.1.429 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.

25.1.430 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.

25.1.431 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.

25.1.432 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.

25.1.433 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.

25.1.434 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.

25.1.435 C5003: Unknown array-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

25.1.436 C5004: Unknown struct-union-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

25.1.437 C5005: PACE illegal type

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your support.

25.1.438 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

25.1.439 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

25.1.440 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

25.1.441 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

25.1.442 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.

25.1.443 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.

25.1.444 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

25.1.445 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

25.1.446 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

25.1.447 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

25.1.448 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

25.1.449 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

25.1.450 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

25.1.451 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

25.1.452 C5650: Too many locations, try to split up function

[FATAL]

Description

The function is too large so the internal data structure of the compiler cannot support it.

Tip

Split up the function that causes the message.

See Also

Limitations

25.1.453 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

}

\endcode
```

```
\section tips Tips

Review the code and initialize local variables at their
declaration (e.g. local_i = 0).
```

25.1.454 C5661: Not all control paths return a value

[WARNING]

Description

There are one or more control paths without a return 'value' statement. Your function does not return a value in every case.

Example

```
int glob;

int test(void) {

  if (glob) {

    return 1;

  }

}
```

Tips

Do return a value in all control flows.

25.1.455 C5662: Generic Inline Assembler Error <string>

[ERROR]

Description

An error occurred during intermediate code generation for inline assembly instructions. The particular error is described in textual form.

Example

The inline assembly code taking the address of a local variable, which is not allocated on the stack.

25.1.456 C5663: Instruction or runtime call <Instruction> produces <Size> bytes of code

[WARNING]

Description

The compiler generated an especially slow code pattern. The actual limit is CPU dependent.

Tips

See if the C code can be rewritten so that faster code is generated.

See if powers of two can be used for dividends.

Try to use unsigned types.

Try to use smaller types.

25.1.457 C5664: Instruction or runtime call <Instruction> executes in <Cycles> cycles

[DISABLED]

Description

The compiler generated an especially slow code pattern. The actual limit is CPU dependent.

Tips

- See if the C code can be rewritten so that faster code is generated.
- See if powers of two can be used for dividends.
- Try to use unsigned types.
- Try to use smaller types.

25.1.458 C5680: HLI Error <Detail>

[DISABLE]

Description

This is a generic error message issued by the HLI.

25.1.459 C5681: Unexpected Token

{WARNING]

Description

The HLI issues this message for unexpected tokens during parsing.

25.1.460 C5682: Unknown fixup type

{ERROR]

Description

The HLI issues this message if the parser finds an unknown fixup type.

Tips

If this error occurs, check the list of available fixup types and make sure there are no spelling errors.

25.1.461 C5683: Illegal directive <Ident> ignored

{WARNING]

Description

The HLI issues this message for directives it knows of, but which are ignored.

25.1.462 C5684: Instruction or label expected

[ERROR]

Description

The HLI issues this message if the HLI parser expects an instruction (or directive) or a label, but finds something else. This error shows up if the assembly line is syntactically wrong, or if the assembly instruction or directive is not known.

25.1.463 C5685: Comment or end of line expected

[ERROR]

Description

The HLI issues this message if it finds unexpected tokens after the operands of an assembly instructions. This may be due to an error in the operand syntax or because the comment is not properly delimited.

25.1.464 C5686: Undefined label <Ident>

[ERROR]

Description

The inline assembler issues this message if an undeclared ident is used but not defined as a label. Note that a spelling error in the name of a variable or register may result in this error as well.

Example

In the following code, the label loop is not defined

```
void example (void){

  asm {

    cmpwi  r3,0

    bne    loop

    jne    loop

  }

}
```

25.1.465 C5687: Label <Ident> defined more than once

[ERROR]

Description

The HLI issues this message for labels which are defined more than once within the same function.

25.1.466 C5688: Illegal operand

[ERROR]

Description

The HLI issues this message if an operand is syntactically correct, but it is out of range, or is not constant, the fixup type is not appropriate, etc.

Tip

This error message is relatively unspecific about the particular error. If the reason for this error is unclear, we suggest that you write the same instruction with simpler operands to find out what caused the error.

25.1.467 C5689: Constant or object expected

[ERROR]

Description

The HLI issues this message if the HLI parser expects a constant or an object, but finds another token.

25.1.468 C5690: Illegal fixup for constant

[ERROR]

Description

The HLI issues this message if a fixup for a constant is not possible or the constant is out of range.

25.1.469 C5691: Instruction operand mismatc

[ERROR]

Description

The HLI issues this message if an assembly instruction is not available. Note that the assembly instruction is still known and may be available for other operands.

Tip

This error message is relatively unspecific. It shows up if the type of operands does not match (e.g., a register instead of an immediate) or the number of operands is different, or the overall operand syntax is wrong. If the error is not obvious, we suggest that you find the error by comparison with a corresponding instruction, but with plain assembly operands.

25.1.470 C5692: Illegal fixup

[ERROR]

Description

The HLI issues this message if a particular fixup is illegal the way it is used.

25.1.471 C5693: Cannot generate code for <Error>

[ERROR]

Description

The compiler can not generate code because of the specified reason.

Possible reasons are:

- code for intrinsic function cannot be generated because of wrong arguments or other preconditions
- the compiler does not support a certain construct. Note: this is a generic message.

25.1.472 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

25.1.473 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

25.1.474 C5702: Local variable '<Variable>' declared in function '<Function>' but not referenced

[INFORMATION]

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.

25.1.475 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".

25.1.476 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.

25.1.477 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.

25.1.478 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.

25.1.479 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<<(j-j);  // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

25.1.480 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.

25.1.481 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.

25.1.482 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.

25.1.483 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.

25.1.484 C5907: Addition replaced with shift

[INFORMATION]

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.

25.1.485 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.

25.1.486 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.

25.1.487 C5910: Label removed

[INFORMATION]

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.

25.1.488 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).

25.1.489 C5912: Code in 'if' and 'else' part are the same

[INFORMATION]

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.

25.1.490 C5913: Conditions of 'if' and 'else if' are the same

[INFORMATION]

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.

25.1.491 C5914: Conditions of 'if' and 'else if' are inverted

[INFORMATION]

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.

25.1.492 C5915: Nested 'if' with same conditions

[INFORMATION]

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.

25.1.493 C5916: Nested 'if' with inverse conditions

[INFORMATION]

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.

25.1.494 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.

25.1.495 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.

25.1.496 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."

25.1.497 C5920: Optimize library function <function>

[INFORMATION]

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.

See also

- Option -OiLib

25.1.498 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 (>&gt;), 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 >> 9;  // uch only has 8 bits, warning here



                   // will be optimized to 'res = uch>>7'
```

See also

- Option -Ont

25.1.499 C6000: Creating Asm Include File <file>

[INFORMATION]

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

25.1.500 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 if the file exists and is locked by another application

See also

- Option -La
- Create Assembler Include Files
- pragma CREATE_ASM_LISTING

25.1.501 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

25.1.502 C10000: Nothing more expected

[ERROR]

Description

At the end of an assembly instruction additional text was found.

Example

```
  MOVI R1, 1  1
```

Tips

Use a C style comment to start comments.

Check if it is an illegal or unknown expression.

25.1.503 C10010: Symbol reserved

[ERROR]

Description

Usage of a reserved symbol to define a label.

Example

```
 R0:  ...


 R15: ...
```

Tip

Use an other identifier.

25.1.504 C10011: ':' expected to delimit label

[ERROR]

Description

The identifier must be followed by a colon to define a label.

Example

```
 aLabel ...
```

Tips

Add a colon at the end of the identifier.

25.1.505 C10012: Label multiply defined

[ERROR]

Description

The label is already defined.

Example

```
  aLabel ...


    ...


  aLabel: ...
```

Tips

Choose an other identifier for the label.

25.1.506 C10013: Label '%s' is undefined

[ERROR]

Description

The referenced label is not defined.

Example

```
  BR notDefined
```

```
...
```

Tips

Define this label or reference to an other one.

25.1.507 C10020: Illegal constant value

[ERROR]

Description

An integer is expected.

Example

```
MOVI R1, 1 + 2.3
```

Tips

Use an integral expression.

25.1.508 C10021: ')' expected to close parenthesis

[ERROR]

Description

A close parenthesis is expected to have a valid expression.

Example

```
MOVI R1, (1 + 2 * 3
```

Tips

Set a close parenthesis at the end of the expression.

25.1.509 C10022: Constant expression expected

[ERROR]

Description

A constant expression is expected.

Example

```
MOVI R1, (1 + 2 * / 3)
```

Tips

Use a valid constant expression.

25.1.510 C10023: '(' expected to open parenthesis

[ERROR]

Description

An open parenthesis must be used to define the cast with a C type.

Example

```
DC.W * int 0x1000
```

Tips

Add an open parenthesis.

25.1.511 C10024: Size is not valid

[ERROR]

Description

The specified size is not a valid one.

Example

```
DC.W * (void) 0x1000
```

Tips

Use a valid size.

25.1.512 C10025: Cast size is too large

[ERROR]

Description

The specified size is too large.

Example

```
DC.W * (double) 0x1000
```

Tips

Use a smaller size.

25.1.513 C10026: Undefined field

[ERROR]

Description

The specified field does not exist in the corresponding structure.

Example

```
struct {
  int aField1, aField2;

} aStructure;

...

DC.W aStructure.aField3
```

Tips

Use a defined field.

25.1.514 C10027: Identifier expected

[ERROR]

Description

An identifier is expected to define a field of a structure.

Example

```
  DC.W aStructure.1
```

Tips

Use a defined field.

25.1.515 C10028: Corrupted array

[ERROR]

Description

The value used to index the array is too large.

Example

```
int anArray[3];


...


DC.W anArray[5]
```

Tips

Use a smaller value as index.

25.1.516 C10029: ']' expected to close brackets

[ERROR]

Description

A close bracket is expected to specify a valid access to an array.

Example

```
DC.W anArray[1 + 2
```

Tips

Set a close bracket at the end of the expression.

25.1.517 C10030: Offset out of Object

[WARNING]

Description

The value used to offset the object is too large.

Example

```
int anArray[3];

...

DC.W anArray:15
```

Tips

- Use a smaller value as offset.
- This message can be set as Warning or message with the message move options and pragmas.

25.1.518 C10031: C type is expected

[ERROR]

Description

A valid C type is expected.

Example

```
DC.W * (signed float) 0x1000
```

Tip

Use a valid C type.

25.1.519 C10032: Integer type is expected

[ERROR]

Description

A integral C type is expected.

Example

```
DC.W * (const short signed) 0x1000
```

Tip

Use a integral C type.

25.1.520 C10033: '*' expected to have generic pointer

[ERROR]

Description

A star symbol is expected to define a generic pointer.

Example

```
DC.W * (void) 0x1000
```

Tip

Add a star symbol.

25.1.521 C10034: C or user type expected

[ERROR]

Description

A C type or an user type is expected.

Example

```
DC.W * (aVariable) 0x1000
```

Tip

Use a valid type.

25.1.522 C10035: C object is expected

[ERROR]

Description

A C object is expected.

Example

```
  DC.W aLabel.aField
```

Tip

Use a valid object.

Index

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWMCURS08CMPREF
Revision 10.6, 01/2014