HC(S)12 Compiler Manual

Revised: 15 March 2007

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

I

Overview

The HC(S)12 Compiler manual describes the Compiler used for the Freescale 8-bit MCU (Microcontroller Unit) chip series. This document contains these major sections:

- Overview (this section): Description of the structure of this document and a bibliography of C language programming references
- Using the Compiler: Description of how to run the Compiler
- ANSI-C Library Reference: Description on how the Compiler uses the ANSI-C library
- Appendices: FAQs, Troubleshooting, and Technical Notes

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

II

Using the Compiler

This section contains eleven chapters in the use and operation of the Compiler:

1

Introduction

This chapter describes the Compiler used for the Freescale HC(S)12. The Compiler consists of a **Frontend,** which is language-dependent and a **Backend** that depends on the target processor, the HC(S)12.

The major sections of this chapter are:

Compiler environment

The Compiler can be used as a transparent, integral part of the CodeWarrior Development Studio. Using the CodeWarrior IDE is the recommended way to get your project up and running in minimal time. Alternatively, the Compiler can still be configured and used as a standalone application as a member of a suite of other Build Tool Utilities such as a Linker, Assembler, EPROM Burner, Simulator or Debugger, etc.

A linux version of the HC(S)12 compiler (chc12) runs on Red Hat Linux 9.0. Refer to Appendix D “Using the Linux Command Line Compiler” for more information.

In general, a Compiler translates source code such as from C source code files (`*.c`) and header (`*.h`) files into object-code (`*.o`) files for further processing by a Linker. The `*.c` files contain the programming code for the project’s application, and the `*.h` files have data that is specifically targeted to a particular CPU chip or are interface files for functions. The Compiler can also directly generate an absolute (`*.abs`) file that the Burner uses to produce an S-Record (`*.s19` or `*.sx`) File for programming ROM memories.

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

Project directory

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

In the process of designing a project, you can either start from scratch by making your own project configuration (`*.ini`) file and various layout files for your project for use with standalone project-building tools. On the other hand, you can let CodeWarrior coordinate and manage the entire project. Or, you can begin the construction of your project with CodeWarrior and also use the standalone build tools (Assembler, Compiler, Linker, Simulator/Debugger, etc.) that are included with the CodeWarrior suite.

NOTE The Build Tools are located in the `prog` folder in the CodeWarrior installation. The default location is:
`C:\Program Files\Freescale\CW for HC12\prog`.

Editor

You can associate an editor, including the editor that is integrated into CodeWarrior, with the Compiler to enable both error or positive feedback. You can use the *Configuration* dialog box to configure the Compiler to select your choice of editors when using the Build Tools. Please refer to the Editor Settings dialog box section of this manual.

Using CodeWarrior to manage a project

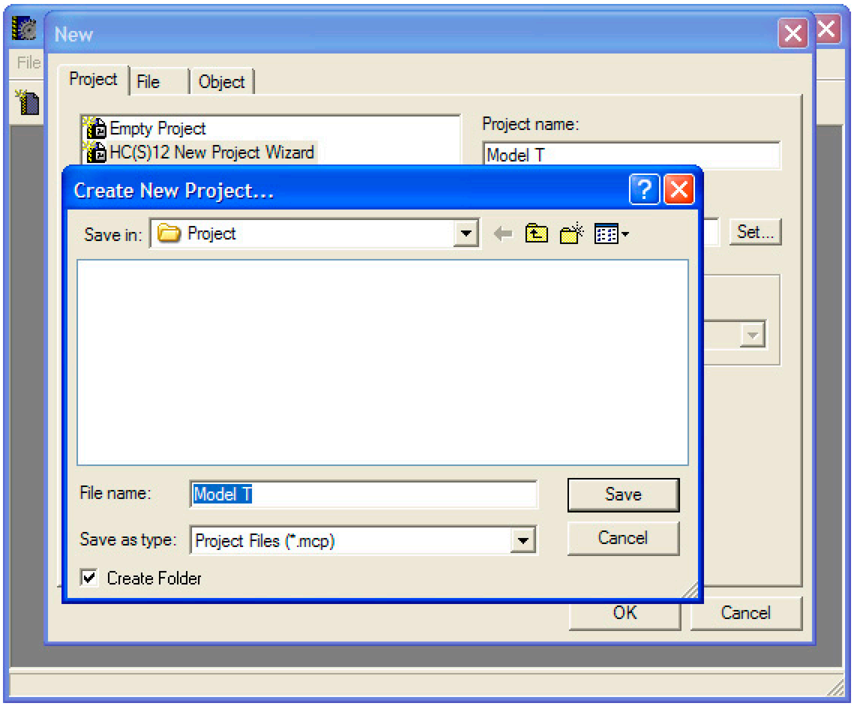
CodeWarrior has a New Project Wizard to easily configure and manage a project. You can get your project up and running by following a short series of steps to configure the project and to generate the basic files which are located in the project directory.

The following New Project Wizard section will construct and configure a basic CodeWarrior project that uses C source code.

New Project Wizard

Start the HC(S)12 CodeWarrior IDE (usual path: `Freescale\CodeWarrior for HC12\bin\IDE.exe`) and select *New...* from the File menu (*File* > *New...*). The *New* dialog box appears (Figure 1.1).

Figure 1.1 Constructing a new CodeWarrior project

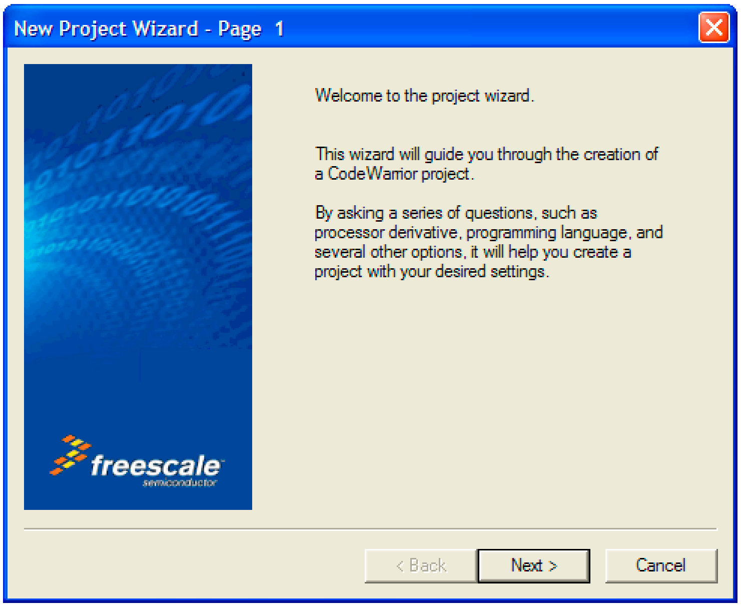


Select the *HC(S)12 New Project Wizard*. Enter the name for your project in the *Project Name* text box. CodeWarrior uses the default `*.mcp` extension automatically, so you do not have to explicitly append the extension to the filename.

In the event that the default location in the *Location* textbox is not where you want to place the project directory, press the *Set* button to the right of the *Location* textbox and browse to the location of your choice in the *Create New Project* dialog box. Check the *Create Folder* checkbox, unless you already prepared another folder. Press the *Save* and the *OK* buttons to close the dialog boxes.

The *New Project Wizard - Page 1* dialog box appears. (Figure 1.2).

Figure 1.2 New Project Wizard - Page 1 dialog box



New Project Wizard - Page 1

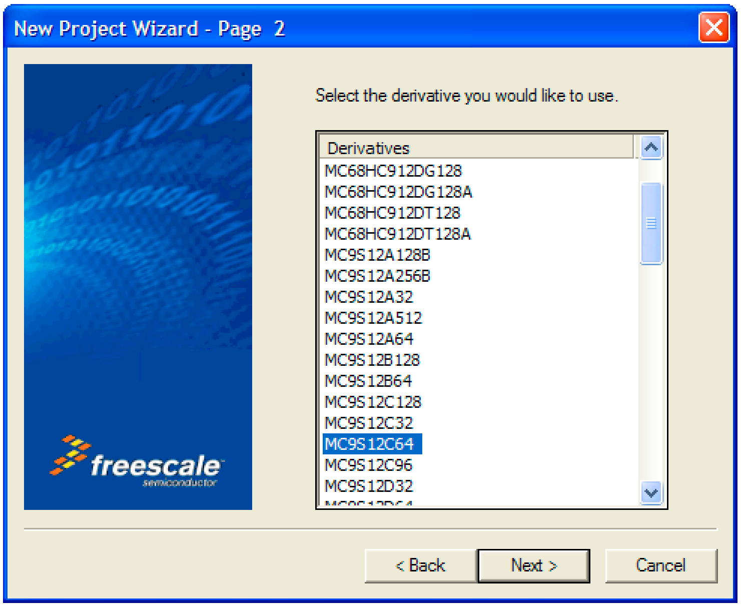
Welcome to the project wizard.

This wizard will guide you through the creation of a CodeWarrior project.

By asking a series of questions, such as processor derivative, programming language, and several other options, it will help you create a project with your desired settings.

The *New Project Wizard* welcomes you aboard. Moving right along... Press *Next >*. The *New Project Wizard - Page 2* dialog box appears (Figure 1.3).

Figure 1.3 New Project Wizard - Page 2 dialog box

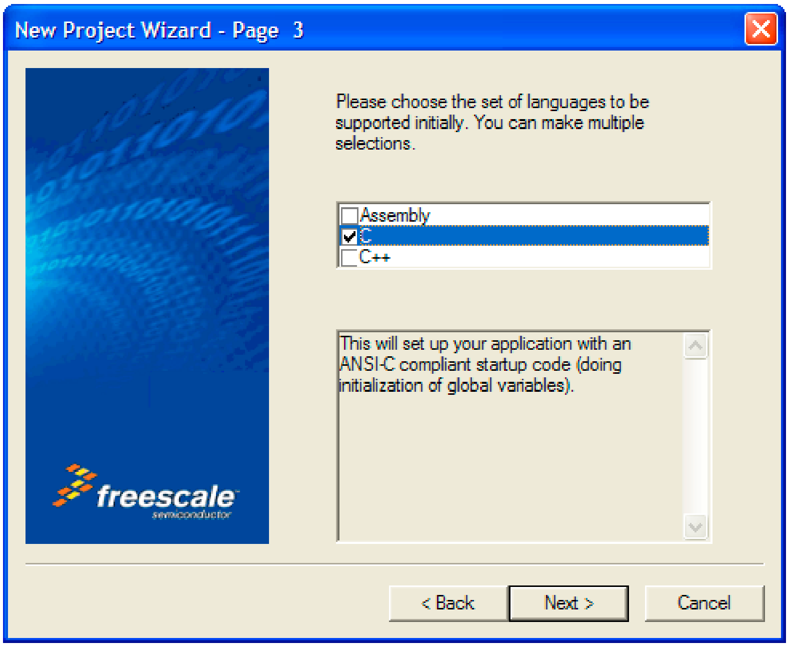

New Project Wizard - Page 2

Select the derivative you would like to use.

```
Derivatives
MC68HC912DG128
MC68HC912DG128A
MC68HC912DT128
MC68HC912DT128A
MC9S12A128B
MC9S12A256B
MC9S12A32
MC9S12A512
MC9S12A64
MC9S12B128
MC9S12B64
MC9S12C128
MC9S12C32
MC9S12C64
MC9S12C96
MC9S12D32
```

Select the desired HC(S)12 CPU derivative for the project. In this case, the *MC9S12C64* is selected. Press *Next >*. The *New Project Wizard - Page 3* dialog box appears (Figure 1.4).

Figure 1.4 New Project Wizard - Page 3 dialog box



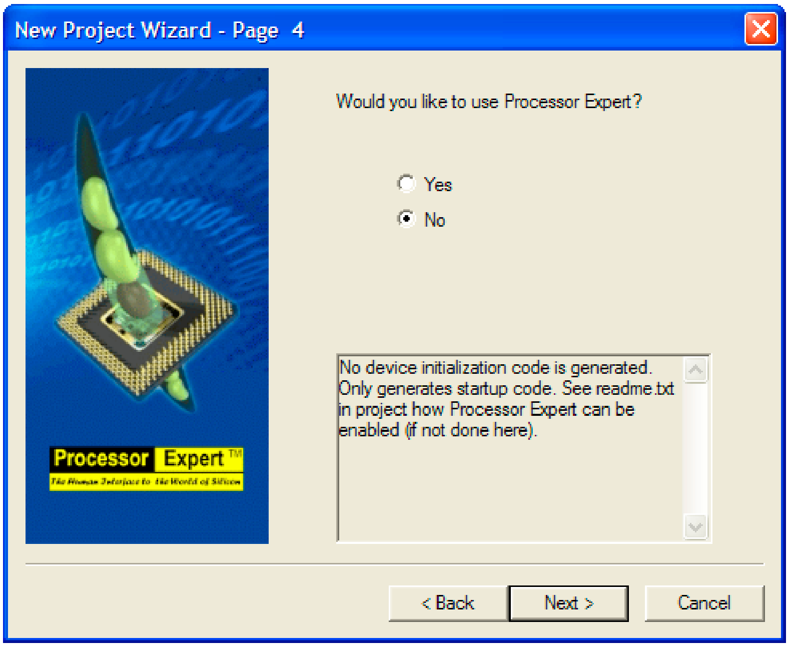
Please choose the set of languages to be supported initially. You can make multiple selections.

- [ ] Assembly
- [✓] C
- [ ] C++

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

The default - *C* is already checked. Uncheck *Assembly* and *C++*. You can have more than one selection for the language, but the *C* option is the only option we will use. Press *Next >*. The *New Project Wizard - Page 4* dialog box appears (Figure 1.6).

Figure 1.5 New Project Wizard - Page 4


New Project Wizard - Page 4
Processor Expert™
The Human Interface to the World of Silicon
< Back
Next >
Cancel

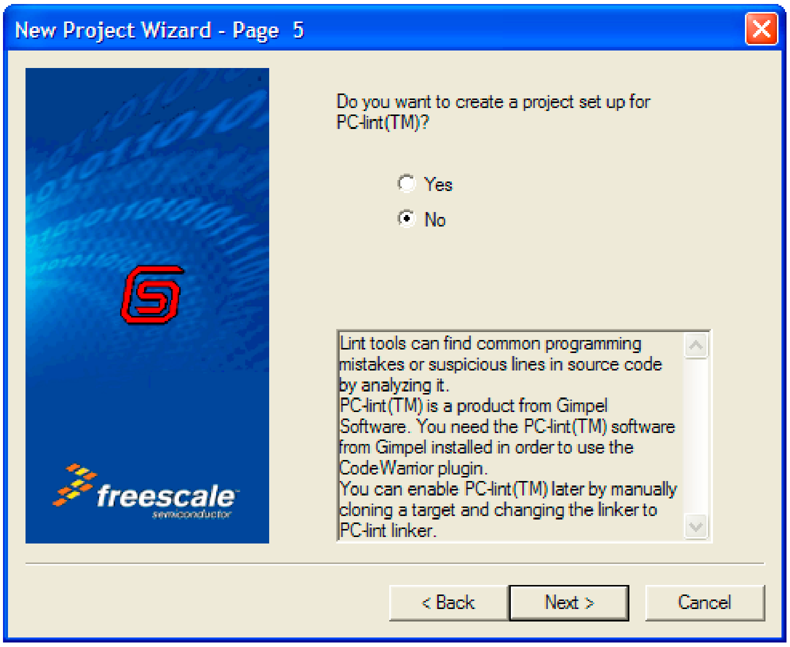
Would you like to use Processor Expert?

Yes [✓ No]

No device initialization code is generated. Only generates startup code. See readme.txt in project how Processor Expert can be enabled (if not done here).

Select *No*. We are interested in creating a simple, basic ANSI-C project. *Processor Expert* is the Rapid Application Development (RAD) tool in the CodeWarrior Development Studio. In practice, you may routinely use Processor Expert on account of its many advantages. The *New Project Wizard - Page 5* dialog box appears (Figure 1.6).

Figure 1.6 New Project Wizard - Page 5 dialog box



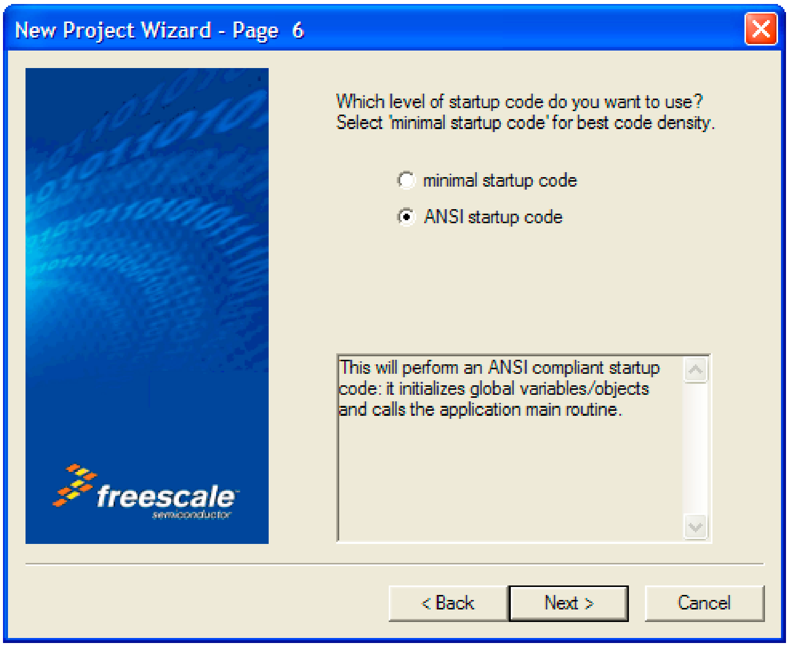
Do you want to create a project set up for PC-lint(TM)?

Yes [✓ No]

Lint tools can find common programming mistakes or suspicious lines in source code by analyzing it.
PC-lint(TM) is a product from Gimpel Software. You need the PC-lint(TM) software from Gimpel installed in order to use the CodeWarrior plugin.
You can enable PC-lint(TM) later by manually cloning a target and changing the linker to PC-lint linker.

The default **-** *No* **-** is what we want. PC-lint is a useful software package for detecting programming errors. But we do not use it here, so press *Next >*. The *New Project Wizard - Page 6* dialog box appears (Figure 1.7).

Figure 1.7 New Project Wizard - Page 6 dialog box



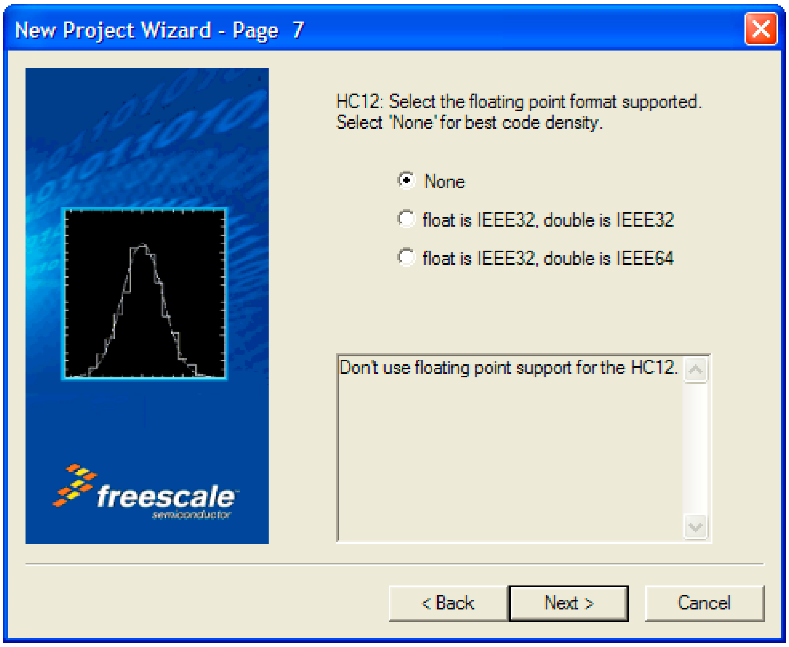
Which level of startup code do you want to use?
Select 'minimal startup code' for best code density.

( ) minimal startup code
(●) ANSI startup code

This will perform an ANSI compliant startup code: it initializes global variables/objects and calls the application main routine.

The default - *ANSI-C startup code* is the usual choice. CodeWarrior automatically generates the startup and initialization routines and calls the entry routine into your ANSI-C project - the `main()` function. Press *Next >*. The *New Project Wizard - Page 7* dialog box appears (Figure 1.8).

Figure 1.8 New Project Wizard - Page 7 dialog box



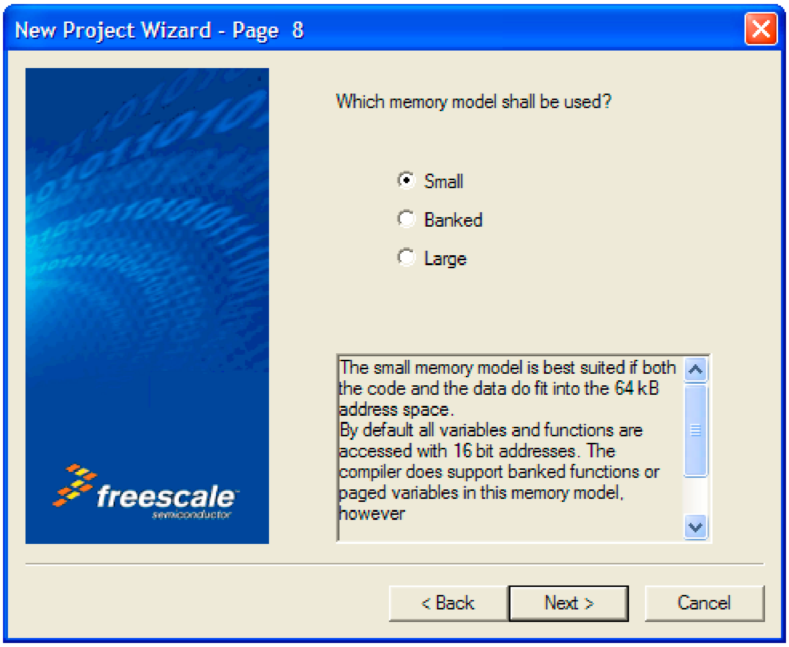
HC12: Select the floating point format supported.
Select 'None' for best code density.

[✓ None]
float is IEEE32, double is IEEE32
float is IEEE32, double is IEEE64

Don't use floating point support for the HC12.

The default - *None* is what you will usually want for your projects. A simple project definitely does not require the complexity of floating-point numbers. Use the integer format whenever possible in your projects, as floating-point numbers impose a severe speed-hit penalty. Press *Next >*. The *New Project Wizard - Page 8* dialog box appears (Figure 1.8).

Figure 1.9 New Project Wizard - Page 8 dialog box



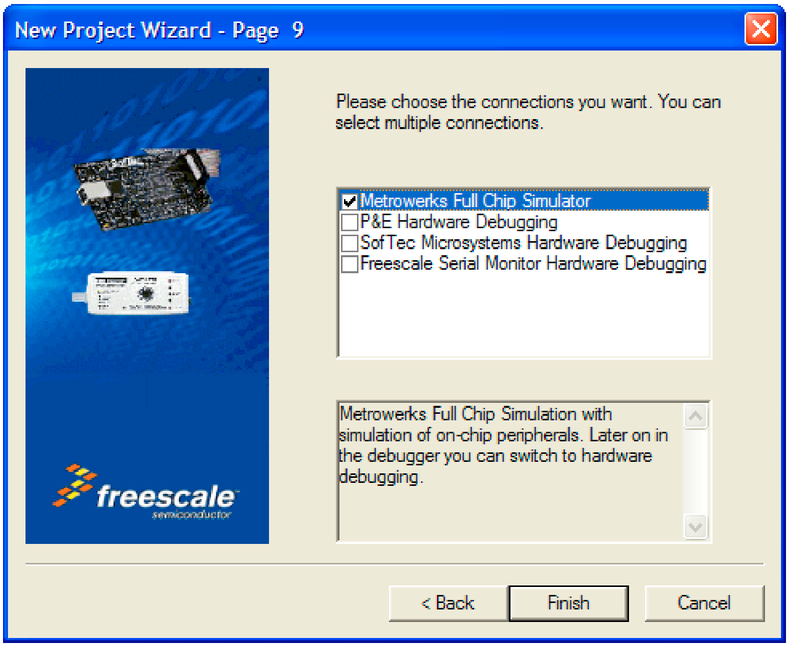
Which memory model shall be used?

(● Small) ○ Banked ○ Large

The small memory model is best suited if both the code and the data do fit into the 64 kB address space.
By default all variables and functions are accessed with 16 bit addresses. The compiler does support banked functions or paged variables in this memory model, however

Select *Small* for the memory model. The *Small* memory model is used for smaller (address space < 64 kB) HC(S)12 projects. Larger projects can make use of the two other memory models. Press *Next >*. The *New Project Wizard - Page 8* dialog box appears (Figure 1.11).

Figure 1.10 New Project Wizard - Page 9



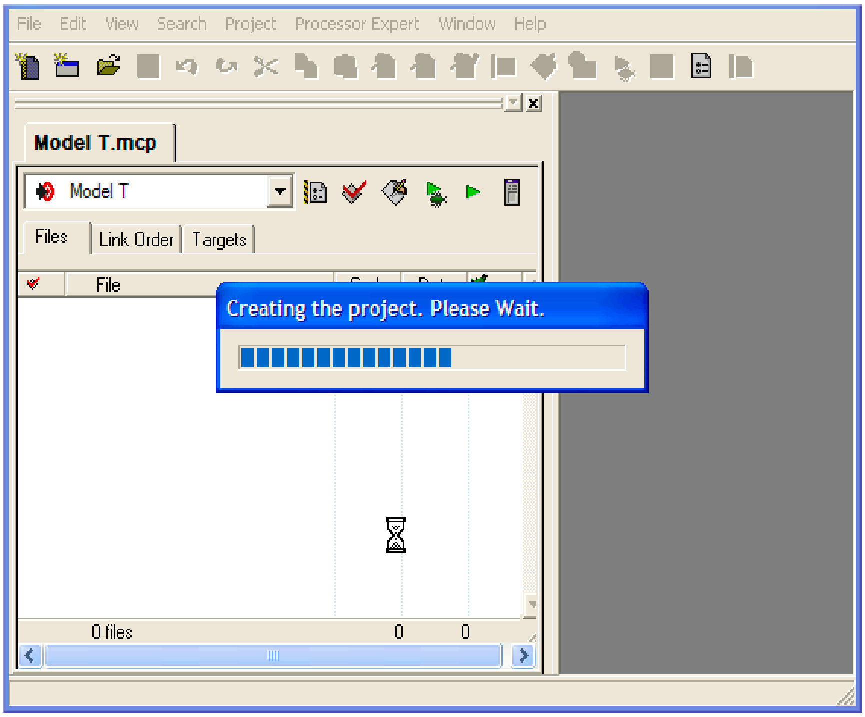
Please choose the connections you want. You can select multiple connections.

- [✓] Metrowerks Full Chip Simulator
- [ ] P&E Hardware Debugging
- [ ] SofTec Microsystems Hardware Debugging
- [ ] Freescale Serial Monitor Hardware Debugging

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

The default - *Full Chip Simulation* (*FCS*) - is our last selection, so press *Finish*. Using the New Project Wizard, an HC(S)12 project could easily be created within a minute or two (Figure 1.11).

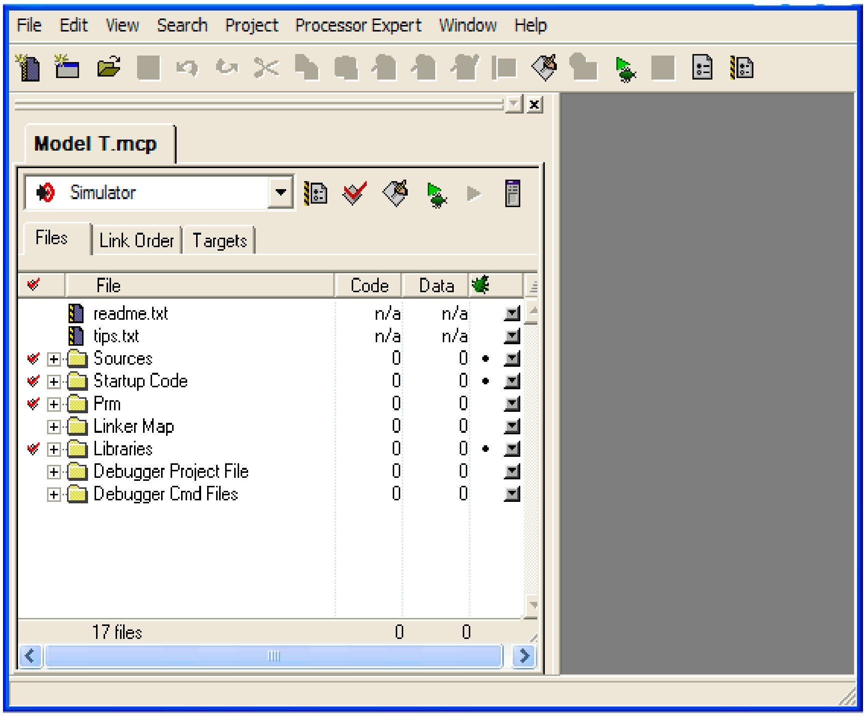
Figure 1.11 Project creation



Creating the project. Please Wait.

CodeWarrior now creates an ANSI-C project (Figure 1.12).

Figure 1.12 CodeWarrior project window



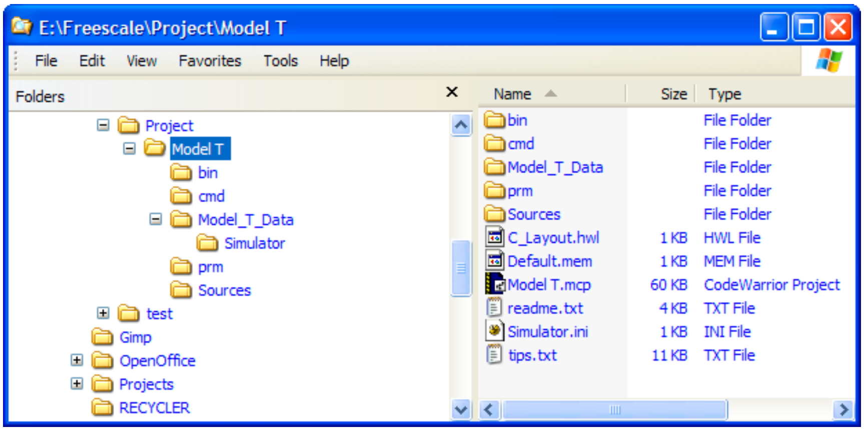
✓	File	Code	Data	
	readme.txt	n/a	n/a	
	tips.txt	n/a	n/a	
✓	Sources	0	0	•
✓	Startup Code	0	0	•
✓	Prm	0	0	
	Linker Map	0	0	
✓	Libraries	0	0	•
	Debugger Project File	0	0	
	Debugger Cmd Files	0	0	
	17 files	0	0	

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

Analysis of the project files and folders

CodeWarrior created a project window that contains two text files and seven “folders.” In reality the folder icons do not necessarily represent any actual folders but instead are convenient groups of project files. If you were to examine with Windows Explorer the project directory that CodeWarrior created for the project, you could view the actual generated project folders and files, as in Figure 1.13. After the final stage of the New Project Wizard, you could safely close the project and return to it later, in the same configuration as when you last saved it.

Figure 1.13 Project directory in the Windows Explorer



Name	Size	Type
bin		File Folder
cmd		File Folder
Model_T_Data		File Folder
prm		File Folder
Sources		File Folder
C_Layout.hwl	1 KB	HWL File
Default.mem	1 KB	MEM File
Model T.mcp	60 KB	CodeWarrior Project
readme.txt	4 KB	TXT File
Simulator.ini	1 KB	INI File
tips.txt	11 KB	TXT File

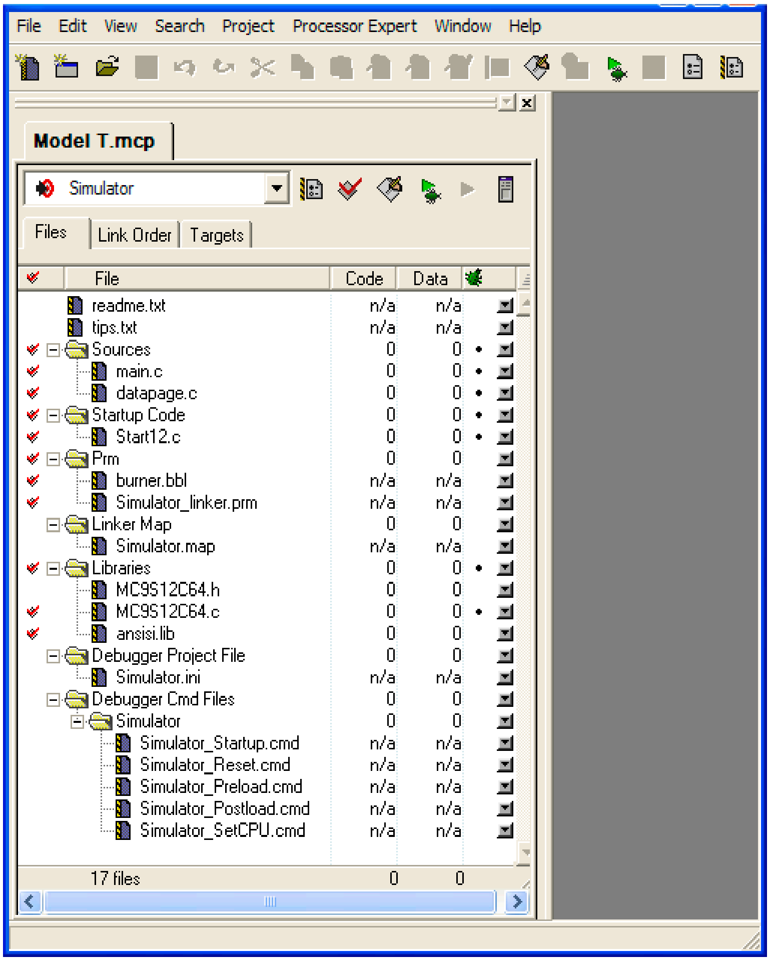
For this project, the name of the project directory and its path is:

```
E:\Freescale\Project\Model T
```

Inside the project directory is the master file for the project - `Model T.mcp`. This is the file that you open whenever you want to reopen the project. Opening this master project file opens the CodeWarrior project in the same configuration it had when it was last saved.

If you expand the “folders” - groups, actually, in the CodeWarrior project window, you can view all the default files that CodeWarrior generated (Figure 1.14).

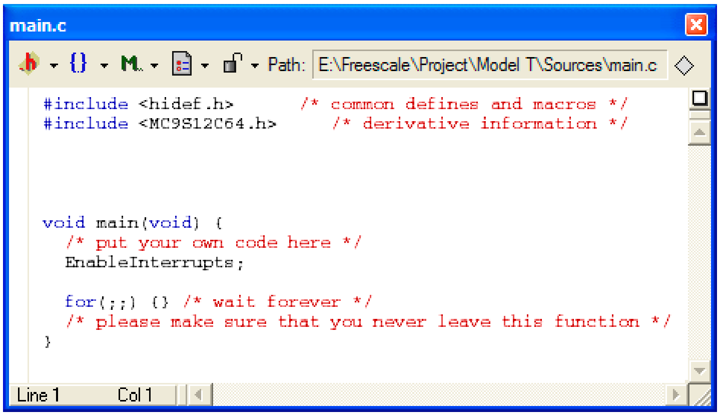
Figure 1.14 Project window showing most of the files that CodeWarrior created



✓	File	Code	Data	
	readme.txt	n/a	n/a	
	tips.txt	n/a	n/a	
✓	Sources	0	0	•
✓	main.c	0	0	•
✓	datapage.c	0	0	•
✓	Startup Code	0	0	•
✓	Start12.c	0	0	•
✓	Prm	0	0	
✓	burner.bbl	n/a	n/a	
✓	Simulator_linker.prm	n/a	n/a	
	Linker Map	0	0	
	Simulator.map	n/a	n/a	
✓	Libraries	0	0	•
	MC9S12C64.h	0	0	
✓	MC9S12C64.c	0	0	•
✓	ansisi.lib	0	0	
	Debugger Project File	0	0	
	Simulator.ini	n/a	n/a	
	Debugger Cmd Files	0	0	
	Simulator	0	0	
	Simulator_Startup.cmd	n/a	n/a	
	Simulator_Reset.cmd	n/a	n/a	
	Simulator_Preload.cmd	n/a	n/a	
	Simulator_Postload.cmd	n/a	n/a	
	Simulator_SetCPU.cmd	n/a	n/a	
	17 files	0	0	

Those files marked by red check marks will remain checked until they are successfully assembled, compiled, or linked. Double click on the `main.c` file in the `Sources` group. The editor in CodeWarrior opens the `main.c` file in the project window that CodeWarrior generated (Figure 1.15).

Figure 1.15 main.c opened in the project window



```
#include <hidef.h>      /* common defines and macros */
#include <MC9S12C64.h>     /* derivative information */



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

  for(;;) {} /* wait forever */
  /* please make sure that you never leave this function */
}
```

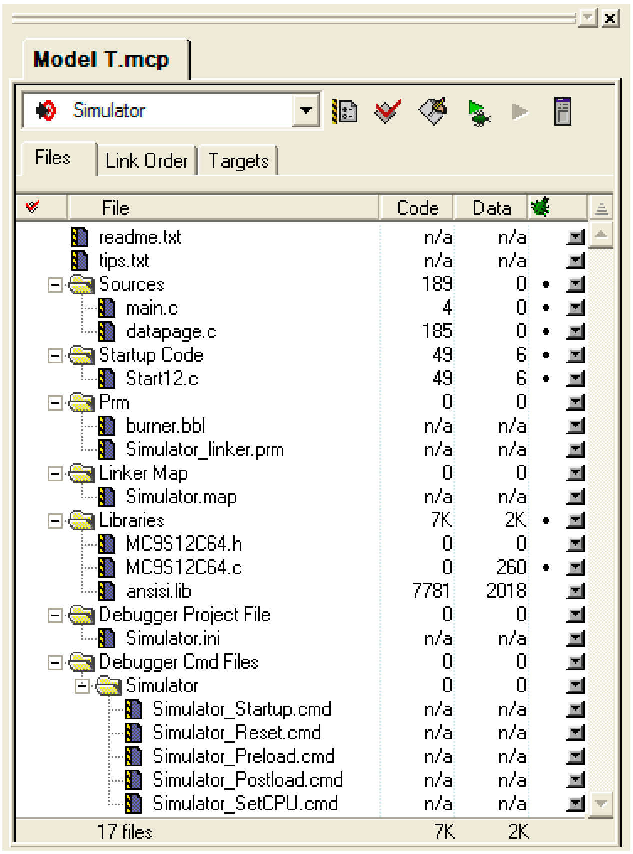
You could adapt the `main.c` file created by the Wizard as a base for your C source program. Otherwise, you can import other C source-code files into the project and remove the default `main.c` file from the project. Whichever way you go concerning the C source code, you need one and only one `main()` function for your project.

For now we will use the simple `main.c` file. At this point, CodeWarrior has created the project, but the source files have not yet been compiled and no object code has been linked into an executable output file. Return to the CodeWarrior project window.

You could process any of the check-marked files individually or a combination of them simultaneously by selecting their icons in the project window. In this case, we will build the entire project all at once. To do this, you could press the *Make* button on the Toolbar in the project window or you can build your project from the *Project* menu in CodeWarrior (*Project* > *Make* (or *Debug*).

If CodeWarrior is correctly configured and if the files do not have any serious errors, all of the red check marks in the project window will disappear after a successful building of the project (Figure 1.16)

Figure 1.16 Successful build of your project



Model T.mcp

Simulator

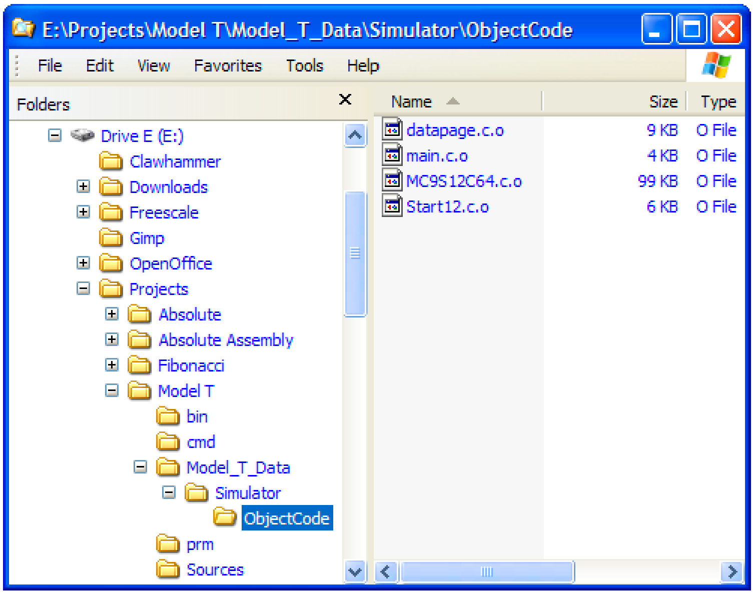
Files | Link Order | Targets

File	Code	Data	
readme.txt	n/a	n/a	
tips.txt	n/a	n/a	
Sources	189	0	•
main.c	4	0	•
datapage.c	185	0	•
Startup Code	49	6	•
Start12.c	49	6	•
Prm	0	0	
burner.bbl	n/a	n/a	
Simulator_linker.prm	n/a	n/a	
Linker Map	0	0	
Simulator.map	n/a	n/a	
Libraries	7K	2K	•
MC9S12C64.h	0	0	
MC9S12C64.c	0	260	•
ansisi.lib	7781	2018	
Debugger Project File	0	0	
Simulator.ini	n/a	n/a	
Debugger Cmd Files	0	0	
Simulator	0	0	
Simulator_Startup.cmd	n/a	n/a	
Simulator_Reset.cmd	n/a	n/a	
Simulator_Preload.cmd	n/a	n/a	
Simulator_Postload.cmd	n/a	n/a	
Simulator_SetCPU.cmd	n/a	n/a	
17 files	7K	2K	

Continually compiling and linking your project files incrementally during the construction phase of the project is a wise programming technique in case an error occurs. The source of the error is much easier to locate if the project is frequently rebuilt. You can make use of the positive or error feedback for each compilation.

This project has four C-source files that successfully compiled. The *Code* and *Data* columns in the project window show the size of the compiled executable object code or the non-executable data in the object code for the compiled source files. Some additional files were generated after the build process (Figure 1.17).

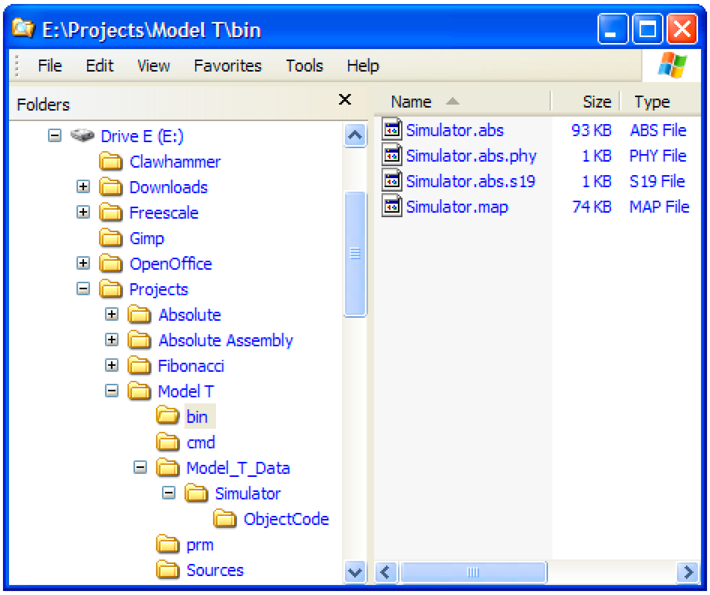
Figure 1.17 Windows Explorer after a project build



Name	Size	Type
datapage.c.o	9 KB	O File
main.c.o	4 KB	O File
MC9S12C64.c.o	99 KB	O File
Start12.c.o	6 KB	O File

The object-code files for the four C-source files are found in the *ObjectCode* folder. However, the executable output file is located in the `bin` folder (Figure 1.18).

Figure 1.18 bin folder in the project directory



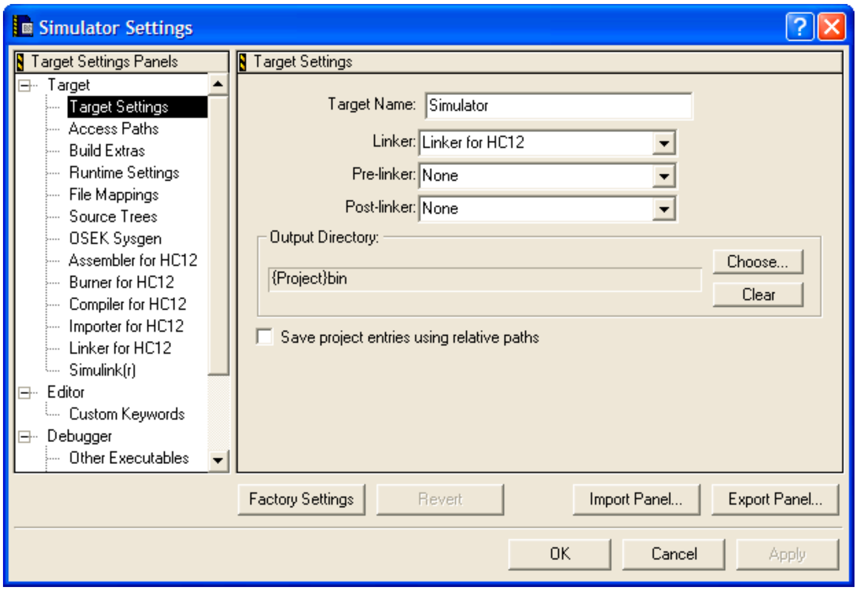
Name	Size	Type
Simulator.abs	93 KB	ABS File
Simulator.abs.phy	1 KB	PHY File
Simulator.abs.s19	1 KB	S19 File
Simulator.map	74 KB	MAP File

As you can see, all the files currently in the `bin` folder have the `Simulator` filename plus an extension. The extension for the executable is `*.abs` (for absolute). The `*.s19` file extension is the *S-Record File* used for programming ROM memory. The `*.map` file extension is for the *Linker Map file*. The Map file provides (among other things) useful information concerning how the Linker allocates RAM and ROM memory areas for the various modules used in the project.

You have not entered these filenames - `Simulator.*` - while creating the project with the *New Project Wizard.* So where did these filenames come from? These so happen to be the default filenames for the project using the New Project Wizard. You can change these defaults to become more meaningful, say *Alpha.**, by using one of the preference panels available in CodeWarrior - *Target Settings*.

From the *Edit* menu in CodeWarrior, select *Edit > Simulator Settings...*. The *Simulator Settings* dialog box appears with the *Target Settings* preference panel (Figure 1.19).

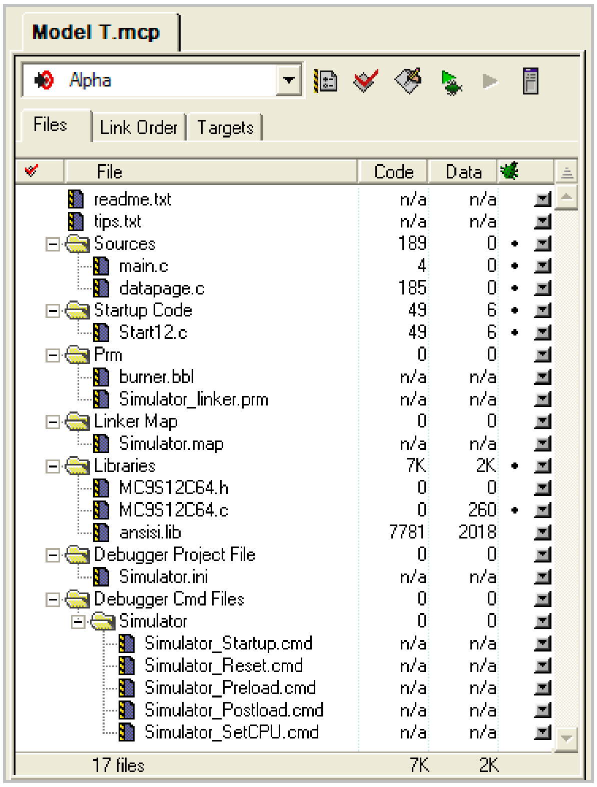
Figure 1.19 Target Settings preference panel



Target Name: Simulator
Linker: Linker for HC12
Pre-linker: None
Post-linker: None
Output Directory: {Project}bin
Save project entries using relative paths

The *Target Name:* text box contains the default *Target Name* for the project. Enter `Alpha` in this text box and press *OK*. If you were to again check the *Edit* menu, you would notice that the *Simulator Settings...* menu item is no longer present, while *Alpha Settings...* is there is its place. This change is also reflected in the project window. *Alpha* now appears as the new name for the build target (Figure 1.20).

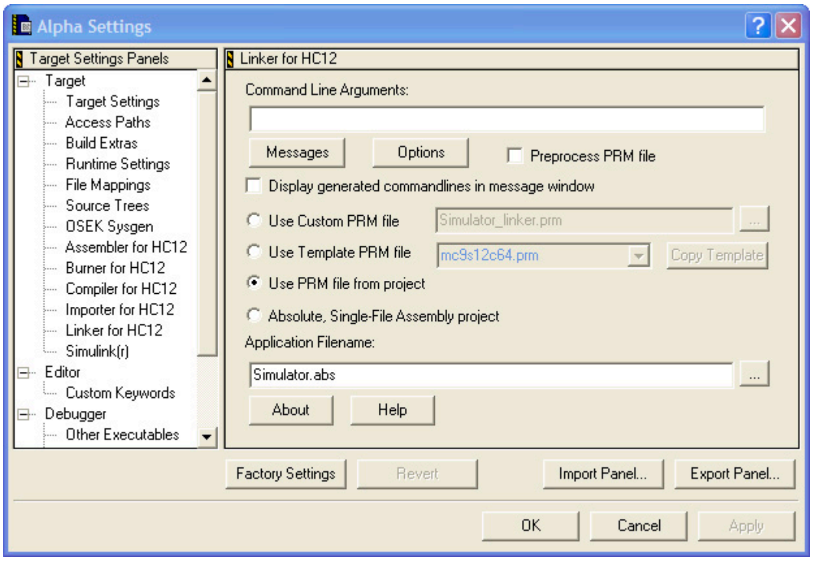
Figure 1.20 Alpha is the new name for the build target...



File	Code	Data	
readme.txt	n/a	n/a	
tips.txt	n/a	n/a	
Sources	189	0	•
main.c	4	0	•
datapage.c	185	0	•
Startup Code	49	6	•
Start12.c	49	6	•
Prm	0	0	
burner.bbl	n/a	n/a	
Simulator_linker.prm	n/a	n/a	
Linker Map	0	0	
Simulator.map	n/a	n/a	
Libraries	7K	2K	•
MC9S12C64.h	0	0	
MC9S12C64.c	0	260	•
ansisi.lib	7781	2018	
Debugger Project File	0	0	
Simulator.ini	n/a	n/a	
Debugger Cmd Files	0	0	
Simulator	0	0	
Simulator_Startup.cmd	n/a	n/a	
Simulator_Reset.cmd	n/a	n/a	
Simulator_Preload.cmd	n/a	n/a	
Simulator_Postload.cmd	n/a	n/a	
Simulator_SetCPU.cmd	n/a	n/a	
17 files	7K	2K	

This causes the name of the `Simulator` folder which contains the object files to be changed to `Alpha`. However, the names in the `bin` folder still are unchanged. You can change the name of the executable file to `Alpha.abs` by using another preference panel. From the *Edit* menu, select *Alpha Settings...*. The *Alpha Settings* dialog box appears. Select *Target > Linker for HC12* in the *Target Settings Panels*. The *Linker for HC12* preference panel appears (Figure 1.21).

Figure 1.21 Linker for HC12 preference panel



Command Line Arguments:

Preprocess PRM file [ ]

Display generated commandlines in message window [ ]

Use Custom PRM file: Simulator_linker.prm

Use Template PRM file: mc9s12c64.prm — Copy Template

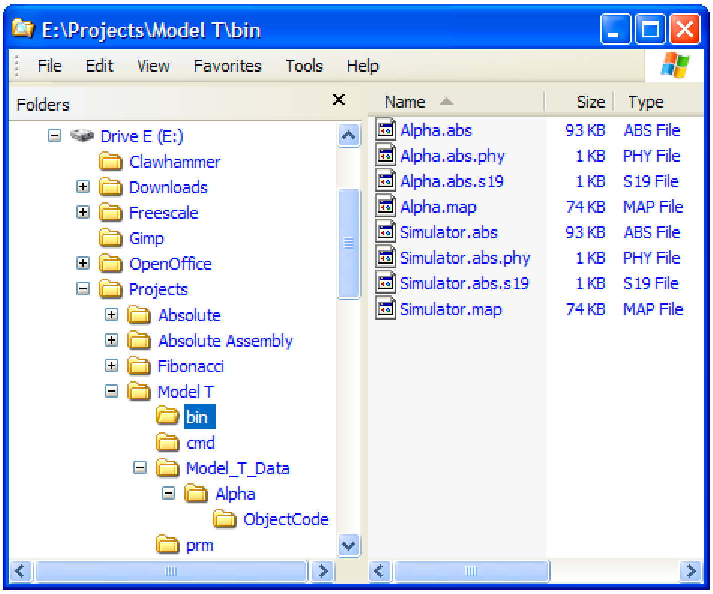
[✓ Use PRM file from project]

Absolute, Single-File Assembly project

Application Filename: Simulator.abs

In the *Application Filename:* text box, delete *Simulator.abs* and enter `Alpha.abs` and press *OK*. Now a dialog box appears stating that "*Target 'Alpha' must be relinked*". Press *OK*. Press the *Make* icon on the Toolbar to rebuild the project. The contents of the `bin` folder change to reflect the new build target - *Alpha* (Figure 1.22).

Figure 1.22 bin folder revisited...



Name	Size	Type
Alpha.abs	93 KB	ABS File
Alpha.abs.phy	1 KB	PHY File
Alpha.abs.s19	1 KB	S19 File
Alpha.map	74 KB	MAP File
Simulator.abs	93 KB	ABS File
Simulator.abs.phy	1 KB	PHY File
Simulator.abs.s19	1 KB	S19 File
Simulator.map	74 KB	MAP File

Now, files with the `Alpha.*` filenames are generated. The previous `Simulator.*` files are not modified at all. However, they no longer are included in the project, so that they may be safely deleted.

The Linker PRM file

The PRM file determines how the Linker allocates the RAM and ROM memory areas. The usual procedure is to use the default PRM file in the project window for any particular CPU derivative. However, it is possible to modify the PRM file if you want an alternative allocation.

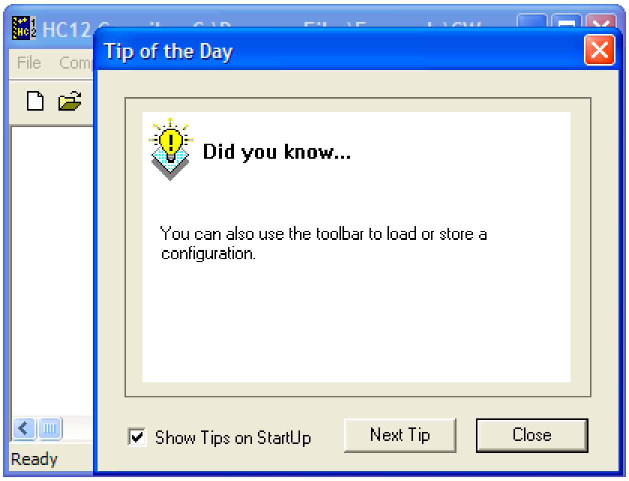
Compilation with the Compiler

It is also possible to use the HC(S)12 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 Project Wizard*. CodeWarrior can create, configure, and mange a project much easier and quicker than using the Build Tools. However, the Build Tools could also create and configure a project from scratch. Instead, we will create a new project directory for this project, but will make use of some files already created in the previous project.

A Build Tool such as the Compiler makes use of a project directory file for configuring and locating its generated files. The folder that is properly configured for this purpose is referred to by a Build Tool as the "*current directory*."

Start the Compiler. You can do this by opening the `chc12.exe` file in the `prog` folder in the HC12 CodeWarrior installation. The Compiler opens (Figure 1.23).

Figure 1.23 HC12 Compiler opens...



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

Read any of the Tips if you choose to and then press *Close* to close the *Tip of the Day* dialog box.

Configuring the Compiler

A Build Tool, such as the Compiler, requires information from configuration files. There are two types of configuration data:

- Global

  This data is common to all Build Tools and projects. There may be common data for each Build Tool (Assembler, Compiler, Linker, ...) such as listing the most recent projects, etc. All tools may store some global data into the `mcutools.ini` file. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no `mcutools.ini` file in this directory, the tool looks for an `mcutools.ini` file located in the MS WINDOWS installation directory (e.g. `C:\WINDOWS`). See Listing 1.1.

Listing 1.1 Typical locations for a global configuration file

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

If a tool is started in the `C:\Program Files\Freescale\CW for HC12 V4.5\prog` directory, the initialization file in the same directory as the tool is used.

`C:\Program Files\Freescale\CW for HC12 V4.5\prog\mcutools.ini).`

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

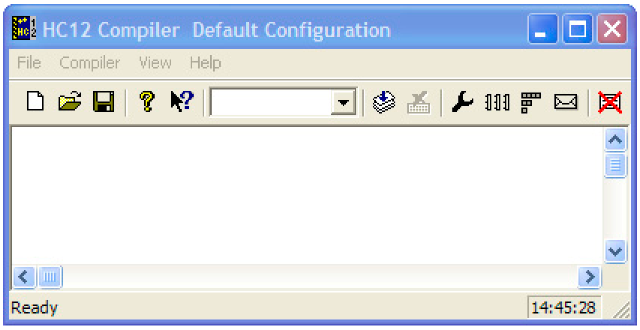
For information about entries for the global configuration file, see Global Configuration-File Entries in the Appendices.

- Local

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

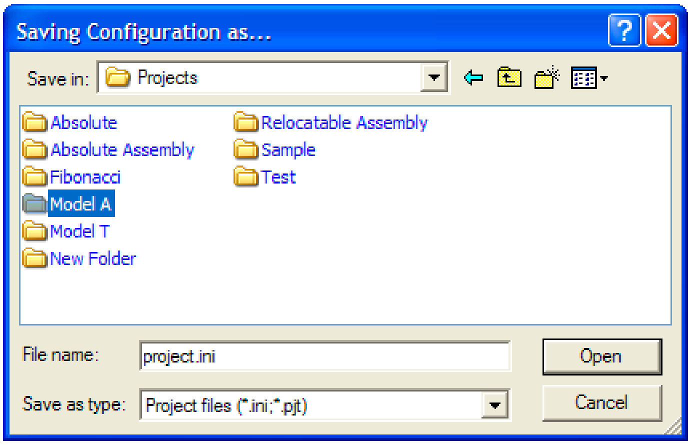
After opening the compiler, you would load the configuration file for your project if it already had one. However, you will create a new configuration file and save it so that when the project is reopened, its previously saved configuration state will be used. From the *File* menu, select *New / Default Configuration*. The *HC12 Compiler Default Configuration* dialog box appears (Figure 1.24)

Figure 1.24 HC12 Compiler Default Configuration dialog box



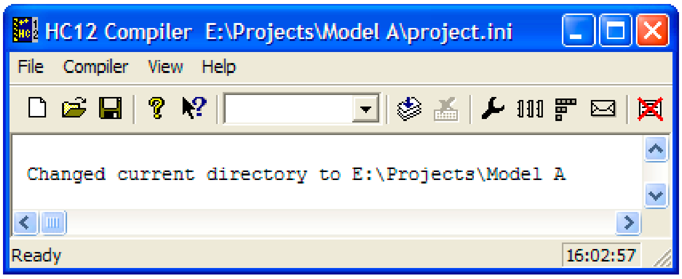
Now save this configuration in a newly created folder that will become the project directory. From the *File* menu, select *Save Configuration* (or *Save Configuration As...*). A *Saving Configuration as...* dialog box appears. Navigate to the folder of your choice and create and name a folder and filename for the configuration file (Figure 1.25).

Figure 1.25 Loading configuration dialog box



Press *Open* and *Save*. The current directory of the HC12 Compiler changes to your new project directory (Figure 1.26).

Figure 1.26 Compiler's current directory switches to your project directory...



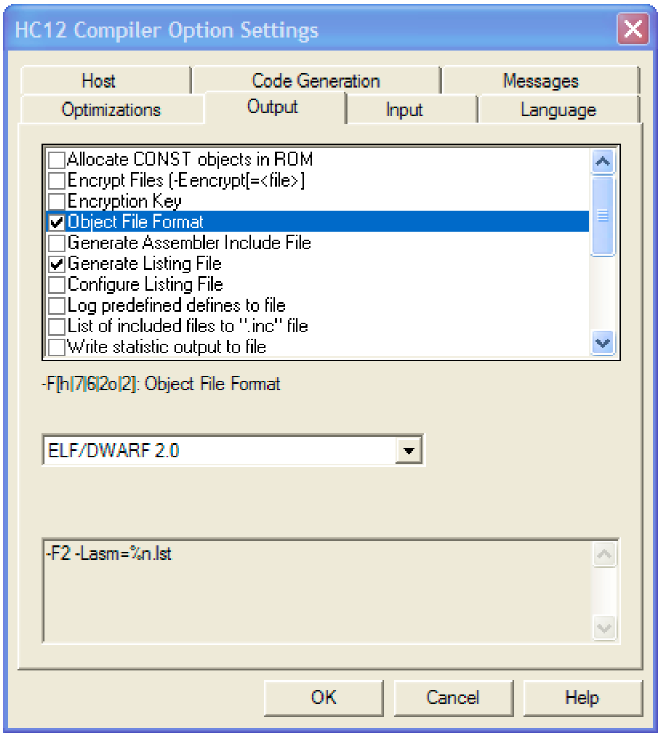
```
Changed current directory to E:\Projects\Model A
```

If you were to examine the project directory with the Windows Explorer at this point, it would only contain the `project.ini` configuration file that you just created. If you further examined the contents of the project's configuration file, you would notice that it now contains the *[CHC12_Compiler]* portion of the `project.ini` file in the `prog` folder where the Build Tools are located. Any options added to or deleted from your project by any Build Tool would be placed into or deleted from this configuration file in the appropriate section for each Build Tool.

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

You now set the object file format that you intend to use (HIWARE or ELF/DWARF). Select the menu entry *Compiler* > *Options...* > *Options*. The Compiler displays the *HC12 Compiler Option Settings* dialog box. Select the *Output* tab (Figure 1.27).

Figure 1.27 HC12 Compiler Option Settings dialog box



- [ ] 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[h|7|6|2o|2]: Object File Format

ELF/DWARF 2.0

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

In the *Output* panel, select the check boxes labeled *Generate Listing File* and *Object File Format*. For the *Object File Format*, select the *ELF/DWARF 2.0* in the pull-down menu. Press *OK* to close the *HC12 Compiler Option Settings* dialog box.

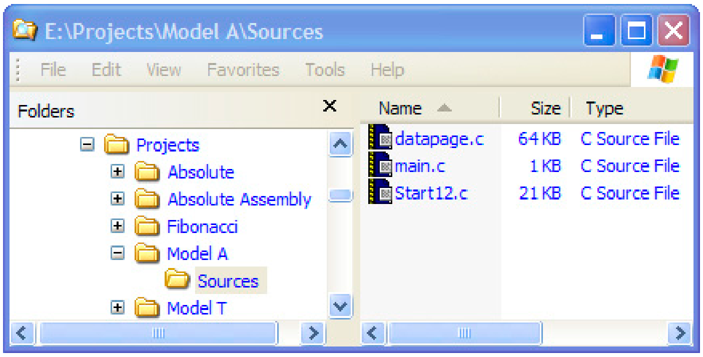
Save the changes to the configuration by:

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

Input Files

Now that the project's configuration is set, you can compile an C source-code file. However, the project does not contain any source-code files at this point. You could create C source `(*.c)` and include `(*.inc)` files from scratch for this project. However, for simplicity's sake, you can copy and paste the Sources folder from the previous Model T CodeWarrior project into the `Model A` project directory (Figure 1.28).

Figure 1.28 Project files



Name	Size	Type
datapage.c	64 KB	C Source File
main.c	1 KB	C Source File
Start12.c	21 KB	C Source File

Now there are four files in the project:

- the `project.ini` configuration file in the project directory and
- in the `Sources` folder:
  - `datapage.c`,

    A collection of paged data-access runtime routines
  - `main.c`, and

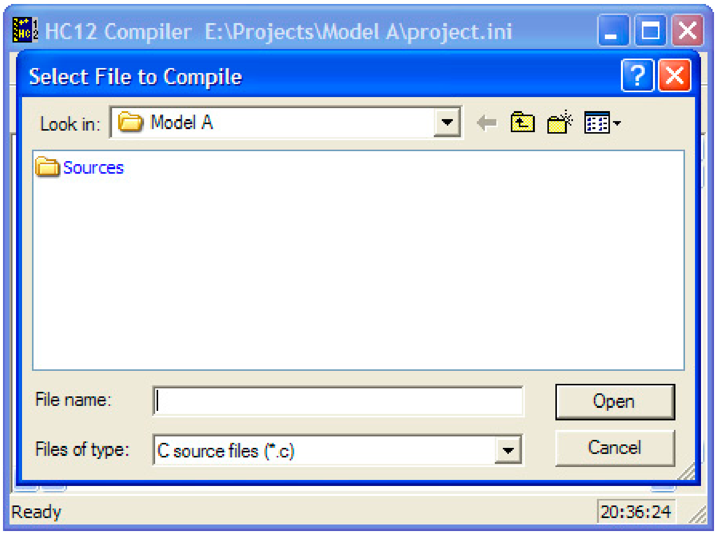
    The user's program plus derivative-specific and memory-model includes
  - `Start12.c`.

    The startup and initialization routines

Compiling the C source-code files

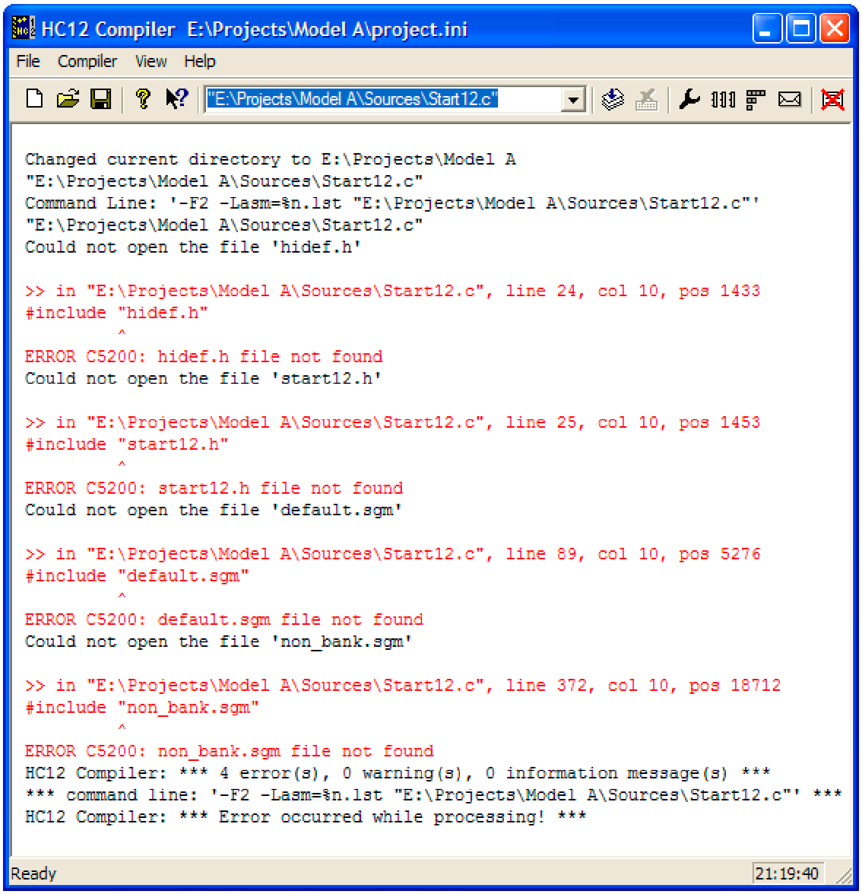
Let's compile one of the C source files, say the `Start12.c` file. From the *File* menu, select *Compile*. The *Select File to Compile* dialog box appears (Figure 1.29).

Figure 1.29 Select File to Compile dialog box


HC12 Compiler E:\Projects\Model A\project.ini
Select File to Compile
Look in: Model A
Sources
File name:
Files of type: C source files (*.c)
Open
Cancel
Ready
20:36:24

Browse to the `Sources` folder in the project directory and select the `Start12.c` file. Press *Open* and the `Start12.c` file should start compiling (Figure 1.30).

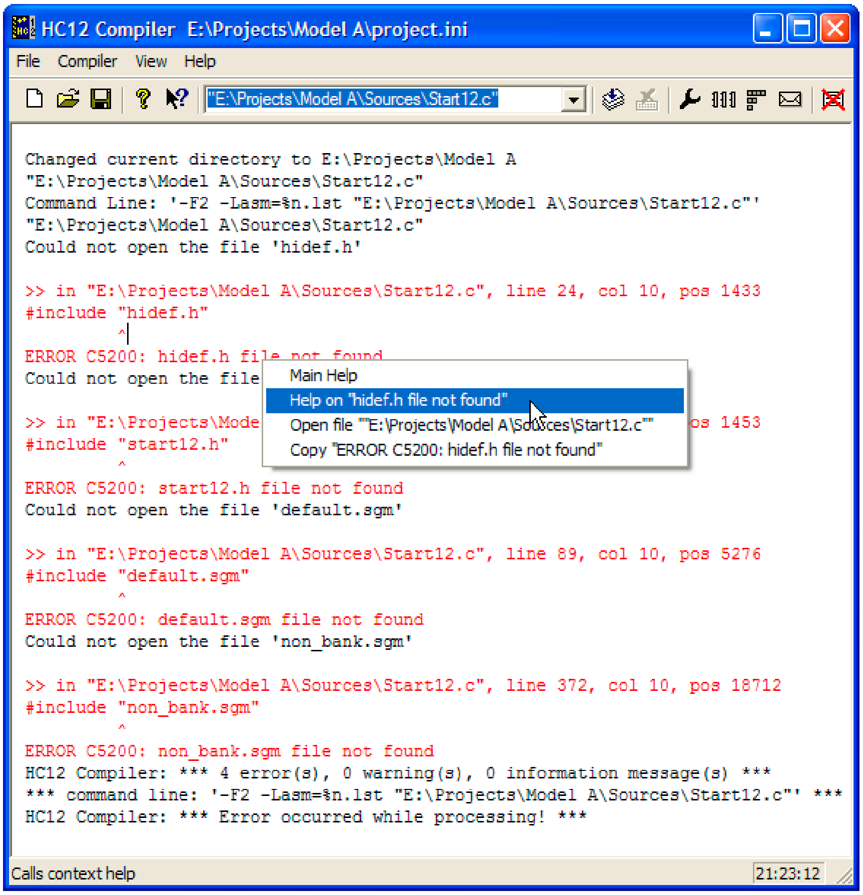
Figure 1.30 Results of compiling the Start12.c file


HC12 Compiler E:\Projects\Model A\project.ini
File Compiler View Help
"E:\Projects\Model A\Sources\Start12.c"
Ready
21:19:40

```
Changed current directory to E:\Projects\Model A
"E:\Projects\Model A\Sources\Start12.c"
Command Line: '-F2 -Lasm=%n.lst "E:\Projects\Model A\Sources\Start12.c"'
"E:\Projects\Model A\Sources\Start12.c"
Could not open the file 'hidef.h'

>> in "E:\Projects\Model A\Sources\Start12.c", line 24, col 10, pos 1433
#include "hidef.h"
         ^
ERROR C5200: hidef.h file not found
Could not open the file 'start12.h'

>> in "E:\Projects\Model A\Sources\Start12.c", line 25, col 10, pos 1453
#include "start12.h"
         ^
ERROR C5200: start12.h file not found
Could not open the file 'default.sgm'

>> in "E:\Projects\Model A\Sources\Start12.c", line 89, col 10, pos 5276
#include "default.sgm"
         ^
ERROR C5200: default.sgm file not found
Could not open the file 'non_bank.sgm'

>> in "E:\Projects\Model A\Sources\Start12.c", line 372, col 10, pos 18712
#include "non_bank.sgm"
         ^
ERROR C5200: non_bank.sgm file not found
HC12 Compiler: *** 4 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst "E:\Projects\Model A\Sources\Start12.c"' ***
HC12 Compiler: *** Error occurred while processing! ***
```

The project window provides positive or negative feedback information about the compilation process or generates error messages if the compiling was unsuccessful. In this case four error messages are generated - four instances of the *C5200: 'FileName' file not found* message. If you right-click on the text about the error message, a context menu appears (Figure 1.31).

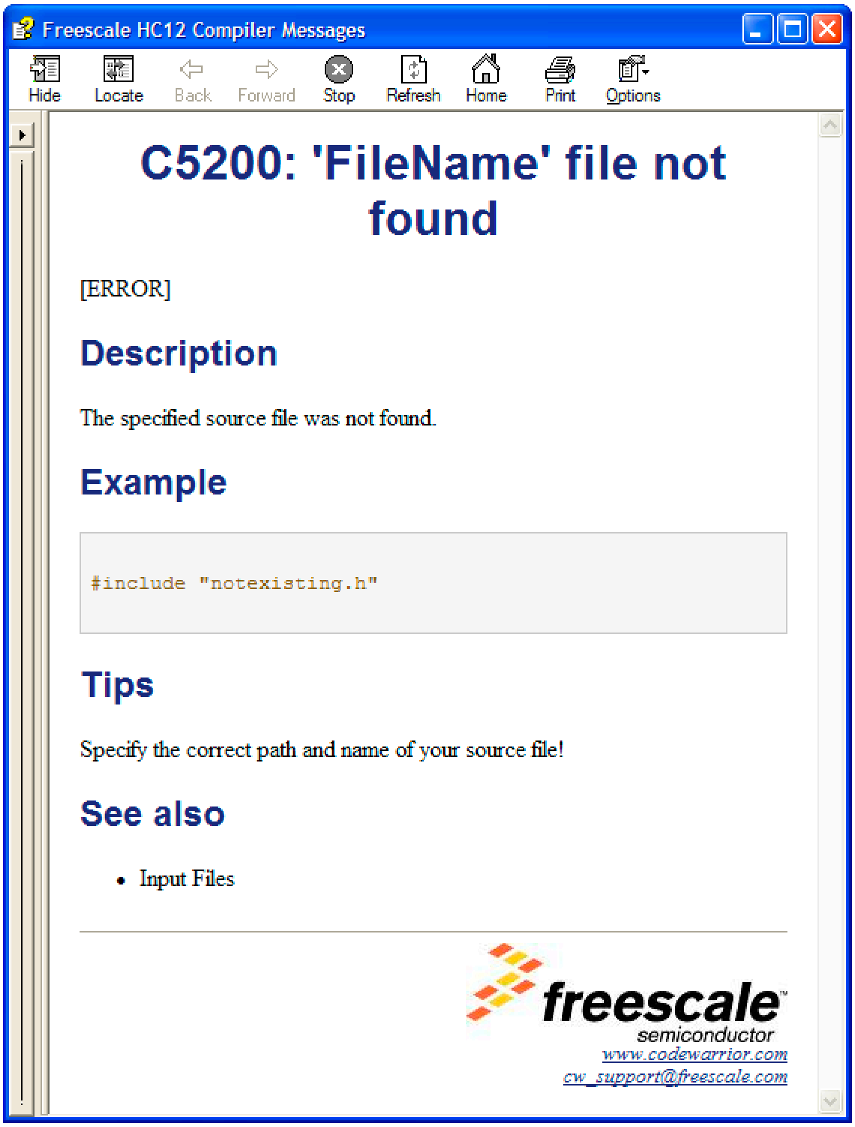
Figure 1.31 Context menu


HC12 Compiler E:\Projects\Model A\project.ini
File Compiler View Help
"E:\Projects\Model A\Sources\Start12.c"
Main Help
Help on "hidef.h file not found"
Open file ""E:\Projects\Model A\Sources\Start12.c""
Copy "ERROR C5200: hidef.h file not found"
Calls context help
21:23:12

```
Changed current directory to E:\Projects\Model A
"E:\Projects\Model A\Sources\Start12.c"
Command Line: '-F2 -Lasm=%n.lst "E:\Projects\Model A\Sources\Start12.c"'
"E:\Projects\Model A\Sources\Start12.c"
Could not open the file 'hidef.h'

>> in "E:\Projects\Model A\Sources\Start12.c", line 24, col 10, pos 1433
#include "hidef.h"
         ^
ERROR C5200: hidef.h file not found
Could not open the file

>> in "E:\Projects\Mode                                                os 1453
#include "start12.h"
         ^
ERROR C5200: start12.h file not found
Could not open the file 'default.sgm'

>> in "E:\Projects\Model A\Sources\Start12.c", line 89, col 10, pos 5276
#include "default.sgm"
         ^
ERROR C5200: default.sgm file not found
Could not open the file 'non_bank.sgm'

>> in "E:\Projects\Model A\Sources\Start12.c", line 372, col 10, pos 18712
#include "non_bank.sgm"
         ^
ERROR C5200: non_bank.sgm file not found
HC12 Compiler: *** 4 error(s), 0 warning(s), 0 information message(s) ***
*** command line: '-F2 -Lasm=%n.lst "E:\Projects\Model A\Sources\Start12.c"' ***
HC12 Compiler: *** Error occurred while processing! ***
```

Select *Help on ‘FileName’ file not found* and help for the C5200 error message appears (Figure 1.32).

Figure 1.32 C5200 error message help



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

freescale semiconductor
www.codewarrior.com
cw_support@freescale.com

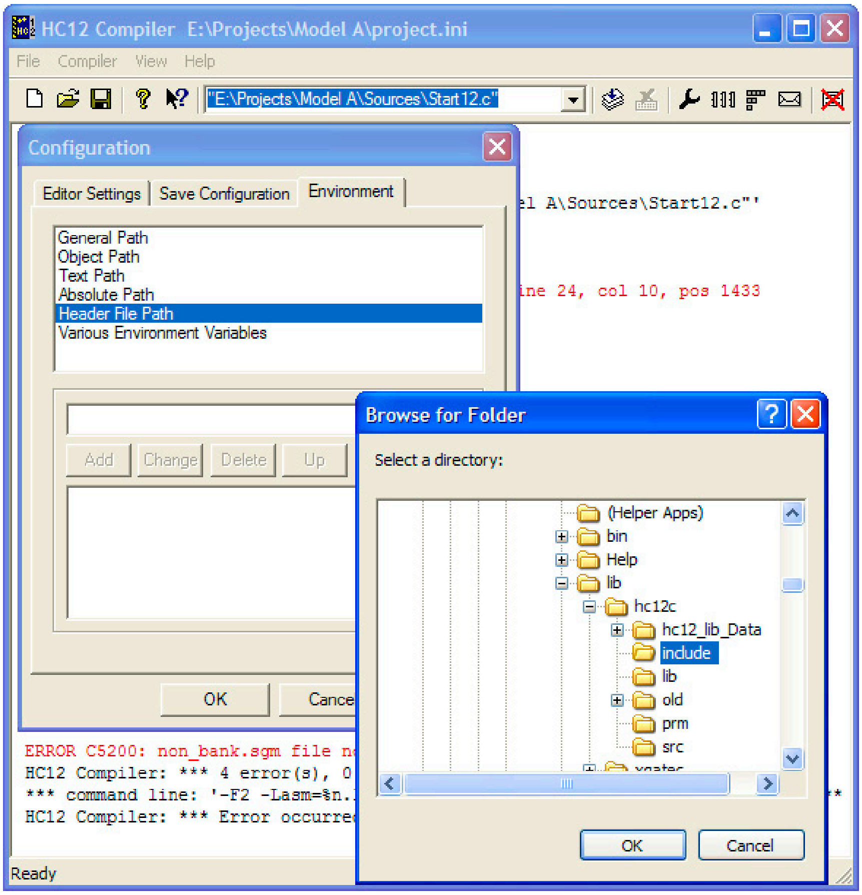
The *Tips* portion in the *Help for the C5200 error* states that you should specify the correct paths and names for the source files. All four of the files that the Compiler could not find are contained in the same following folder:

```
<CodeWarrior installation folder>\lib\hc12\include
```

NOTE If you read the `Start.c` file, you could have anticipated this on account of two `#include` preprocessor directives on lines 24 and 25 for two header files. The remaining two missing files were included by those two header files.

The Compiler needs a configurational modification so that it can find these missing files. Select *File* > *Configuration*. The *Configuration* dialog box appears (Figure 1.33).

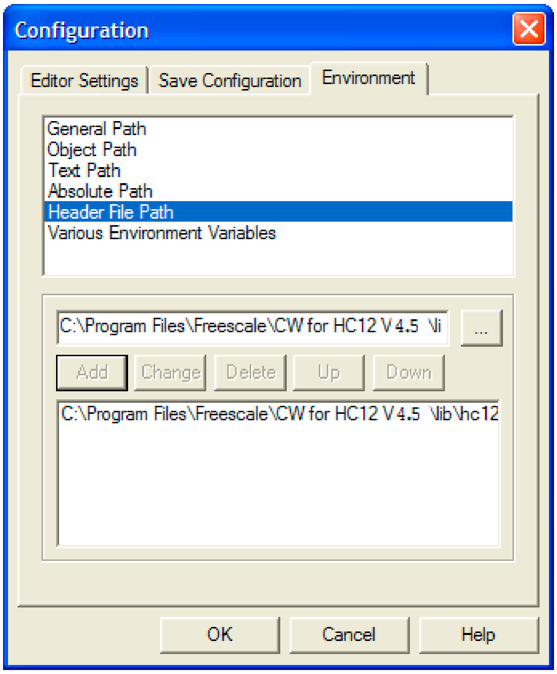
Figure 1.33 Browsing for the include subfolder in the CodeWarrior lib folder



```
el A\Sources\Start12.c"'
ine 24, col 10, pos 1433
ERROR C5200: non_bank.sgm file n
HC12 Compiler: *** 4 error(s), 0
*** command line: '-F2 -Lasm=%n.
HC12 Compiler: *** Error occurre
```

Select the *Environment* tab in the *Configuration* dialog box and then select *Header File Path*. Press the "..." button and navigate in the *Browse for Folder* dialog box for the folder that contains the missing file - the `include` subfolder in the CodeWarrior installation's `lib` folder. Press *OK* to close the *Browse for Folder* dialog box. The *Configuration* dialog box is now again active (Figure 1.34).

Figure 1.34 Adding a Header File Path



C:\Program Files\Freescale\CW for HC12 V 4.5 \li

C:\Program Files\Freescale\CW for HC12 V 4.5 \lib\hc12

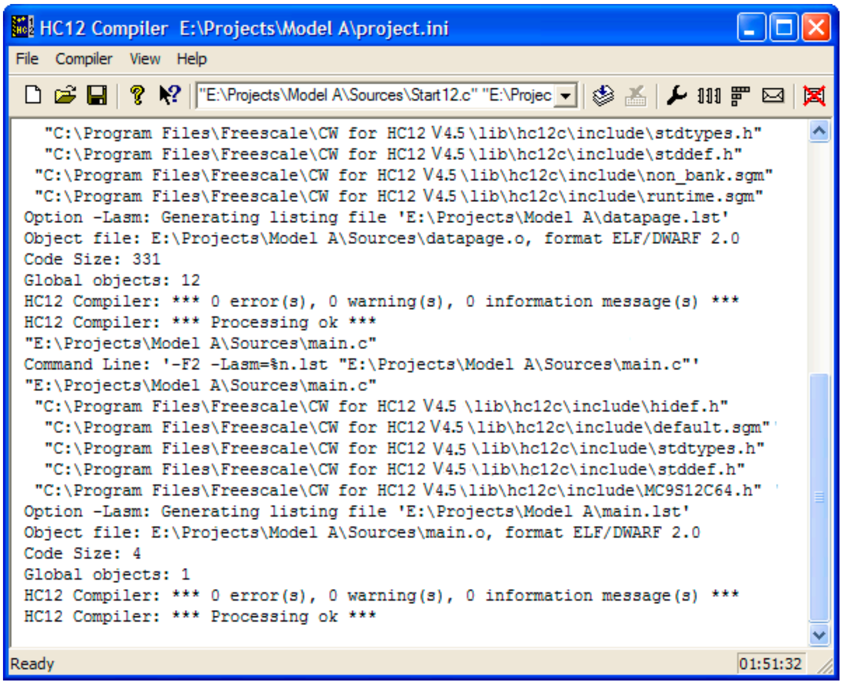
Press the *Add* button. The path to the header files "*C:\Program Files\Freescale\CW for HC12 V5.0\lib\hc12c\include*" now appears in the lower panel. Press *OK*. An asterisk now appears in the *Configuration Title* bar, so save the modification to the configuration by pressing the *Save* button or by *File > Save Configuration*. If you do not save the configuration, the Compiler will revert to last-saved configuration the next time the project is reopened. The asterisk disappears.

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

Now that you have supplied the path to the erstwhile missing files, you can try again to compile the `Start12.c` file. Instead of compiling each file separately, you can compile any or all of them simultaneously.

Select *File > Compile* and again navigate to the Sources folder (in case it is not already active) and this time select all three `*.c` files and press *Open* (Figure 1.35).

Figure 1.35 Successful compilation - three object files created...


HC12 Compiler E:\Projects\Model A\project.ini
File Compiler View Help
"E:\Projects\Model A\Sources\Start12.c" "E:\Projec
Ready
01:51:32

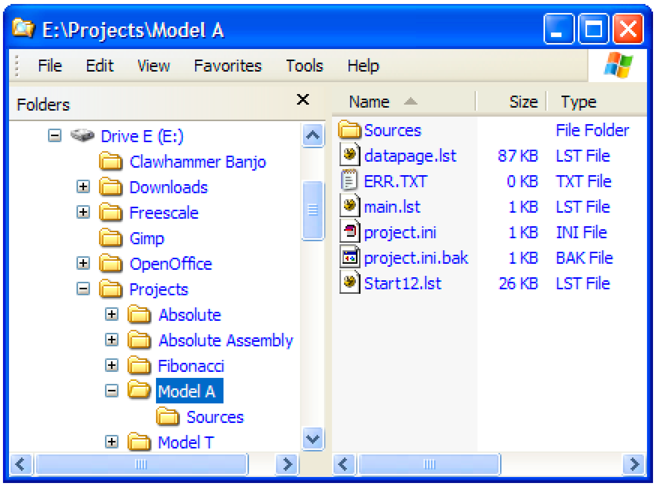
```
    "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\stdtypes.h"
    "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\stddef.h"
   "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\non_bank.sgm"
   "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\runtime.sgm"
  Option -Lasm: Generating listing file 'E:\Projects\Model A\datapage.lst'
  Object file: E:\Projects\Model A\Sources\datapage.o, format ELF/DWARF 2.0
  Code Size: 331
  Global objects: 12
  HC12 Compiler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
  HC12 Compiler: *** Processing ok ***
  "E:\Projects\Model A\Sources\main.c"
  Command Line: '-F2 -Lasm=%n.lst "E:\Projects\Model A\Sources\main.c"'
  "E:\Projects\Model A\Sources\main.c"
   "C:\Program Files\Freescale\CW for HC12 V4.5 \lib\hc12c\include\hidef.h"
    "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\default.sgm"
    "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\stdtypes.h"
    "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\stddef.h"
   "C:\Program Files\Freescale\CW for HC12 V4.5\lib\hc12c\include\MC9S12C64.h"
  Option -Lasm: Generating listing file 'E:\Projects\Model A\main.lst'
  Object file: E:\Projects\Model A\Sources\main.o, format ELF/DWARF 2.0
  Code Size: 4
  Global objects: 1
  HC12 Compiler: *** 0 error(s), 0 warning(s), 0 information message(s) ***
  HC12 Compiler: *** Processing ok ***
```

The Compiler indicates successful compilation of all three C-source files and displays the Code Size for each. Also, the header files included by each C-source file are shown. The message "`*** 0 error(s),`" indicates that the file compiled without errors. Do not forget to save the configuration one additional time.

The Compiler also generated object files in the Sources folder (for further processing by the Linker), and a output listing file in the project directory. The binary object file has the same name as the input module, but with the '`*.o`' extension instead. The assembly output file for each C-source file is similarly named (Figure 1.36).

NOTE The Compiler generates object-code files in the same location as the C-source files. If any C-source code file is in a CodeWarrior library folder (a subfolder inside `\lib`), we recommend that you configure the path for this C-source file into somewhere other than this lib folder. The `OBJPATH` environment variable is used for this case. You use the *Object Path* option in the Configuration dialog box for this (Figure 1.34).

Figure 1.36 Project directory after successful compilation



Name	Size	Type
Sources		File Folder
datapage.lst	87 KB	LST File
ERR.TXT	0 KB	TXT File
main.lst	1 KB	LST File
project.ini	1 KB	INI File
project.ini.bak	1 KB	BAK File
Start12.lst	26 KB	LST File

The haphazard running of this project was intentionally designed to fail in order to illustrate what would occur if the path of any header file is not properly configured. Be aware that header files may be included by C-source or other header files. The `lib` folder in the CodeWarrior installation contains several derivative-specific header and other files available for inclusion into your projects.

Now that the project's object code files are available, the Linker Build Tool (`linker.exe`) together with an appropriate `*.prm` file for the CPU-derivative used in the project could link these object-code files together with any necessary library files to create a `*.abs` executable output file. See the *Linker section in the Build Tool Utilities manual* for details. However, using the CodeWarrior Development Studio is much faster and easier to set up or configure for this purpose.

Linking with the Linker

If you are using the standalone Linker (also known as the *Smart Linker*), you will use a PRM file for the Linker to allocate RAM and ROM memory areas.

- Start your editor and create the project's linker parameter file. You can modify a `*.prm` file from another project and rename it as `<target_name>.prm`.
- Store the PRM file in a convenient location. A good spot would be directly into the project directory.
- In the `<target_name>.prm` file, add the name of the executable (`*.abs`) file, say `<target_name>.abs`. (The actual names chosen for the filenames do not matter, as long as they are unique.) In addition, you can also modify the start and end addresses for the ROM and RAM memory areas. The module's `Model_A.prm` file — a PRM file for an MC9S12C64 from another CodeWarrior project was adapted — is shown in Listing 1.2.

Listing 1.2 Layout of a PRM file for the Linker - Model_A.prm

```
/* This is an adapted linker parameter file for the MC9S12C64 */
LINK Model_A.abs /* This is the name of the executable output file */
NAMES Start12.o datapage.o main.o /* list of all object-code files */
END

SEGMENTS  /* Here all RAM/ROM areas of the device are listed.
             Used in PLACEMENT below. */
    RAM      = READ_WRITE 0x0400   TO 0x0FFF;

    /* unbanked FLASH ROM */
    ROM_4000 = READ_ONLY  0x4000   TO 0x7FFF;
    ROM_C000 = READ_ONLY  0xC000   TO 0xFEFF;

    /* banked FLASH ROM */
    PAGE_3C  = READ_ONLY  0x3C8000 TO 0x3CBFFF;
    PAGE_3D  = READ_ONLY  0x3D8000 TO 0x3DBFFF;
END

PLACEMENT  /* Here all predefined and user segments are placed into
              the SEGMENTS defined above. */

    STARTUP,                        /* startup data structures */
    ROM_VAR,                        /* constant variables */
    STRINGS,                        /* string literals */
    DEFAULT_ROM, NON_BANKED,        /* runtime routines which
                                       must not be banked */
    COPY                            /* copy down information: how to
                                       initialize variables */
                                    /* in case you want to use
                                       ROM_4000 here as well, make sure
                                       that all files (incl. library
                                       files) are compiled with the
                                       option: -OnB=b */
                                    INTO  ROM_C000/*, ROM_4000*/;
    OTHER_ROM                       INTO PAGE_3D, PAGE_3C;

    .stack,                         /* allocate stack first to avoid
                                    overwriting variables on overflow */
    DEFAULT_RAM                     INTO  RAM;
END

STACKSIZE 0x100
VECTOR 0 _Startup /* Reset vector: this is the default
                     entry point for a C/C++ application. */
```

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

NOTE The default size for the stack using the CodeWarrior New Project Wizard for the MC9S12C64 is 256 bytes: (STACKSIZE 0x100).

NOTE Most of the entries in the PLACEMENT section are not used in this simple project. Furthermore, a number of extra entries were deleted from the actual PRM file used in another CodeWarrior project. It does not matter if all of these entries are used or not. They were left in order to show what entries are available for your future projects.

The commands in the linker parameter file are described in detail in the Linker section of the Build Tool Utilities manual.

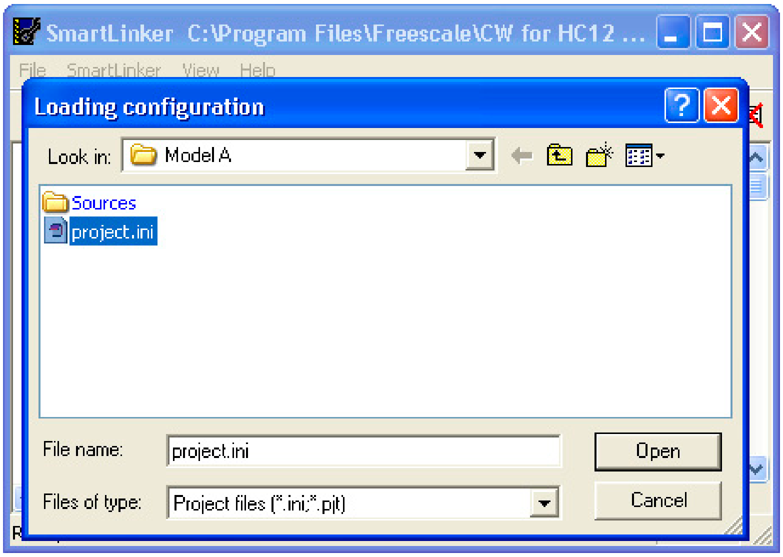
- Start the Linker.

  The Smart Linker tool is located in the `prog` folder in the CodeWarrior installation: `proj\linker.exe`

- Press *Close* to close the *Tip of the Day* dialog box.
- Load the project's configuration file. Use the same `<project>.ini` that the Compiler used for its configuration - the `project.ini` file in the project directory:

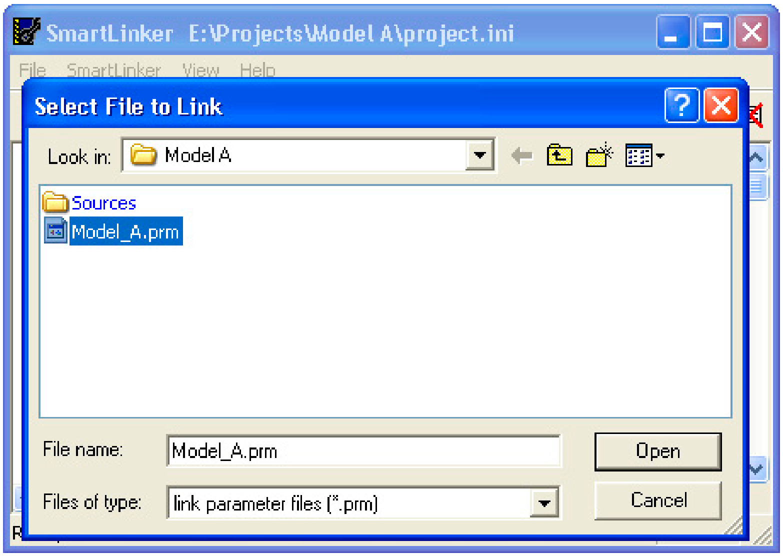
  File > Load Configuration and navigate to the project's configuration file (Figure 1.37).

Figure 1.37 HC(S)12 Linker


SmartLinker C:\Program Files\Freescale\CW for HC12 ...
File SmartLinker View Help
Loading configuration
Look in:
Model A
Sources
project.ini
File name:
project.ini
Open
Files of type:
Project files (*.ini;*.pjt)
Cancel

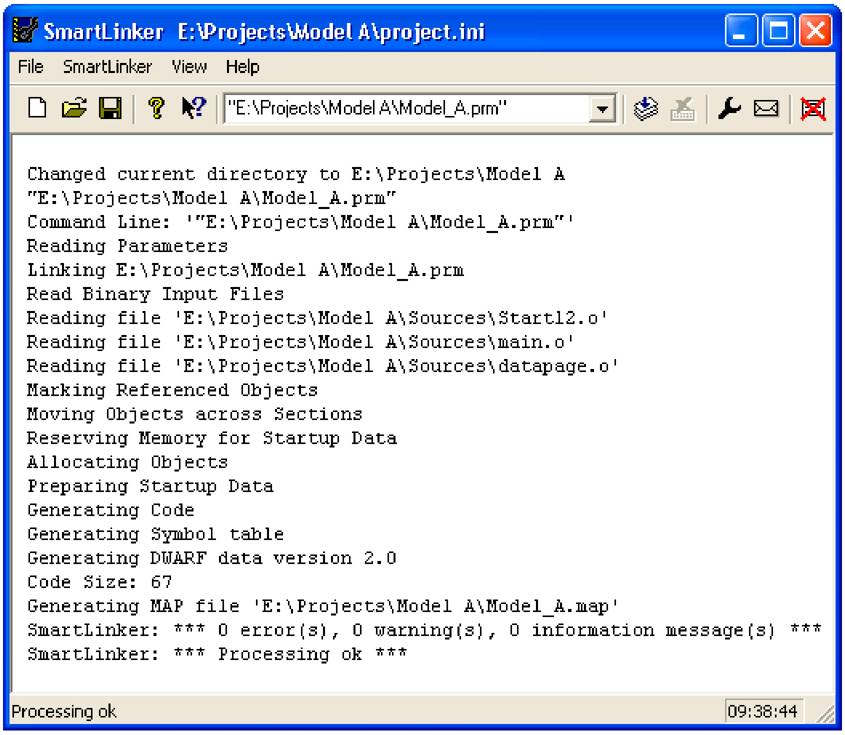
- Press *Open* to load the configuration file. The project directory is now the current directory for the Linker. You can press the *Save* button to save the configuration if you choose. If you fail to save the configuration, the Linker will revert to its last-saved configuration when it is reopened. From the *File* menu in the Smart Linker, select *Link*: (*File* > *Link* (Figure 1.38).

Figure 1.38 Select File to Link dialog box


SmartLinker E:\Projects\Model A\project.ini
File SmartLinker View Help
Select File to Link
Look in:
Model A
Sources
Model_A.prm
File name:
Model_A.prm
Open
Files of type:
link parameter files (*.prm)
Cancel

• Browse to locate the PRM file for your project. Select the PRM file. Press *Open*. The Smart Linker links the object-code files in the `NAMES` section to produce the executable `*.abs` file as specified in the `LINK` portion of the Linker PRM file (Figure 1.39).

Figure 1.39 Linker main window after linking


SmartLinker E:\Projects\Model A\project.ini
File SmartLinker View Help
"E:\Projects\Model A\Model_A.prm"

```
Changed current directory to E:\Projects\Model A
"E:\Projects\Model A\Model_A.prm"
Command Line: '"E:\Projects\Model A\Model_A.prm"'
Reading Parameters
Linking E:\Projects\Model A\Model_A.prm
Read Binary Input Files
Reading file 'E:\Projects\Model A\Sources\Start12.o'
Reading file 'E:\Projects\Model A\Sources\main.o'
Reading file 'E:\Projects\Model A\Sources\datapage.o'
Marking Referenced Objects
Moving Objects across Sections
Reserving Memory for Startup Data
Allocating Objects
Preparing Startup Data
Generating Code
Generating Symbol table
Generating DWARF data version 2.0
Code Size: 67
Generating MAP file 'E:\Projects\Model A\Model_A.map'
SmartLinker: *** 0 error(s), 0 warning(s), 0 information message(s) ***
SmartLinker: *** Processing ok ***
```

The messages in the linker's project window indicate:

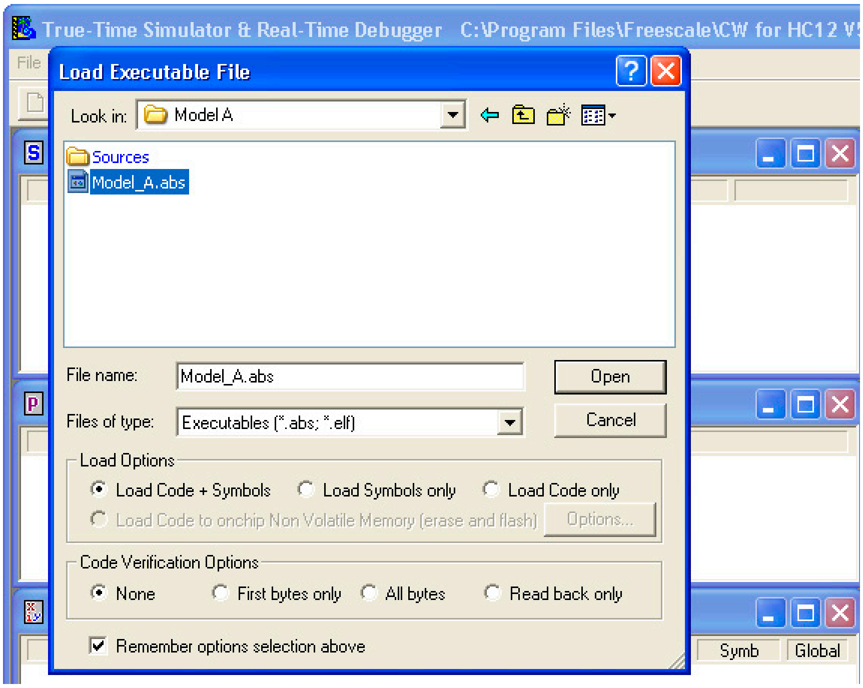
- The current directory for the Linker is the project directory, `E:\Projects\Model A`
- The `Model_A.prm` file was used to name the executable file, which object files were linked, and how the RAM and ROM memory areas are to be allocated for the relocatable sections.
- There were three object-code files, `Start12.o`, `main.o`, and `datapage.o`.
- The output format was DWARF 2.0.
- The Code Size was 67 bytes.
- A Linker Map file was generated - `Model_A.map`.
- No errors or warnings occurred and no information messages were issued.

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

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

    `E:\Projects\Model A\Model_A.abs`

Figure 1.40 HC(S)12 Simulator: Select the executable file


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\CW for HC12 V

Load Executable File

Look in: Model A

Sources
Model_A.abs

File name: Model_A.abs
Files of type: Executables (*.abs; *.elf)

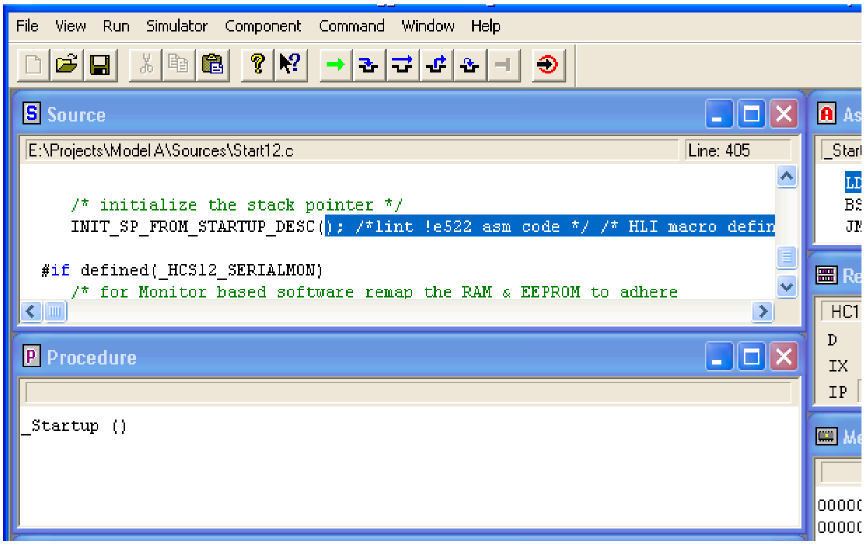
Load Options: [✓ Load Code + Symbols] Load Symbols only, Load Code only, Load Code to onchip Non Volatile Memory (erase and flash)

Code Verification Options: [✓ None] First bytes only, All bytes, Read back only

[✓] Remember options selection above

- Assembly-Step (Figure 1.41) through the program source code (or do something else...).

Figure 1.41 HC(S)12 Simulator Startup



```
E:\Projects\Model A\Sources\Start12.c                    Line: 405

    /* initialize the stack pointer */
    INIT_SP_FROM_STARTUP_DESC(); /*lint !e522 asm code */ /* HLI macro defin

 #if defined(_HCS12_SERIALMON)
    /* for Monitor based software remap the RAM & EEPROM to adhere
```

```
_Startup ()
```

This particular C program could be simulated through its program. You could gain an insight as to what the `Start12.c` routines are before it turns the program over to the routines in `main.c`.

Application Programs (Build Tools)

You will find the standalone application programs (Build Tools) in the \prog directory where you installed the CodeWarrior software. For example, if you installed the CodeWarrior software in the `C:\Program Files\Freescale\` directory, all the Build Tools are located in the `C:\Program Files\Freescale\prog` directory with the exception of `IDE.exe` which is found in the `bin` subfolder of the CodeWarrior installation folder.

The following list is an overview of the tools used for C programming:

- `IDE.exe` - CodeWarrior IDE
- `chc12.exe` - HC(S)12 Compiler
- `ahc12.exe` - HC(S)12 Assembler
- `libmaker.exe` - Librarian Tool to build libraries
- `linker.exe` - Link Tool to build applications (absolute files). The Linker is also referred to as the *Smart Linker*.
- `decoder.exe` - Decoder Tool to generate assembly listings. This is another name for a *Disassembler*.

- `maker.exe` - Make Tool to rebuild automatically
- `burner.exe` - Batch and interactive Burner (S-Record Files, ...)
- `hiwave.exe` - Multi-Purpose Simulation or Debugging Environment
- `piper.exe` - Utility to redirect messages to `stdout`

NOTE Depending on your license configuration, not all programs listed above may be installed or there might be additional programs.

Startup Command-Line Options

There are some special tool options. These tools are specified at tool startup (while launching the tool). They cannot be specified interactively:

- -Prod: Specify Project File at Startup specifies the current project directory or file (Listing 1.3).

Listing 1.3 An example of a startup command-line option

```
linker.exe -Prod=C:\Freescale\demo\myproject.pjt
```

There are other options that launch a build tool and open its special dialog boxes. Those dialog boxes are available in the compiler, assembler, burner, maker, linker, decoder, or libmaker:

- ShowOptionDialog

  This startup option (see Listing 1.4) opens the tool's option dialog box.

- ShowMessageDialog

  This startup option opens the tool message dialog box.

- ShowConfigurationDialog

  This opens the *File* > *Configuration* dialog box.

- ShowBurnerDialog

  This option is for the Burner only and opens the Burner dialog box.

- ShowSmartSliderDialog

  This option is for the compiler only and opens the smart slider dialog box.

- ShowAboutDialog

  This option opens the tool about box.

The above options open a modal dialog box where you can specify tool settings. If you press the OK button of the dialog box, the settings are stored in the current project settings file.

Listing 1.4 An example of storing options in the current project settings file

```
C:\Freescale\prog\linker.exe -ShowOptionDialog
                             -Prod=C:\demos\myproject.pjt
```

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

CodeWarrior Integration

All required plug-ins are installed together with the CodeWarrior IDE. The CodeWarrior IDE is installed in the `'bin'` directory (usually `C:\CodeWarrior\bin`). The plug-ins are installed in the `'bin\plugins'` directory.

Combined or Separated Installations

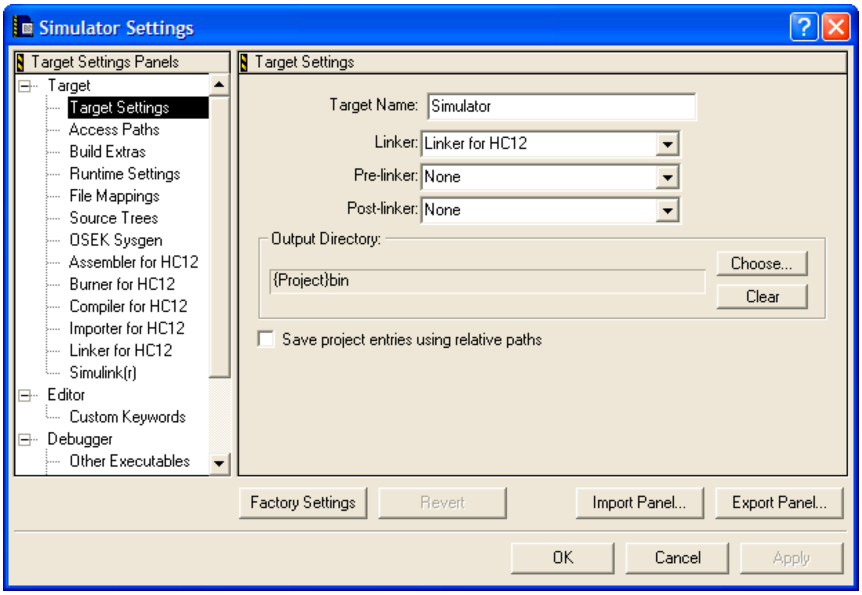
The installation script enables you to install several CPUs in one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

NOTE In addition, 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 the `C:\CodeWarrior\bin`) directory. To do this, start the `regservers.bat` batch file.

Target Settings preference panel

The linker builds an absolute (`*.abs`) file. Before working with a project, set up the linker for the selected CPU in the *Target Settings Preference Panel* (Figure 1.42).

Figure 1.42 Target Settings preference panel



Target Name: Simulator
Linker: Linker for HC12
Pre-linker: None
Post-linker: None
Output Directory: {Project}bin
Save project entries using relative paths

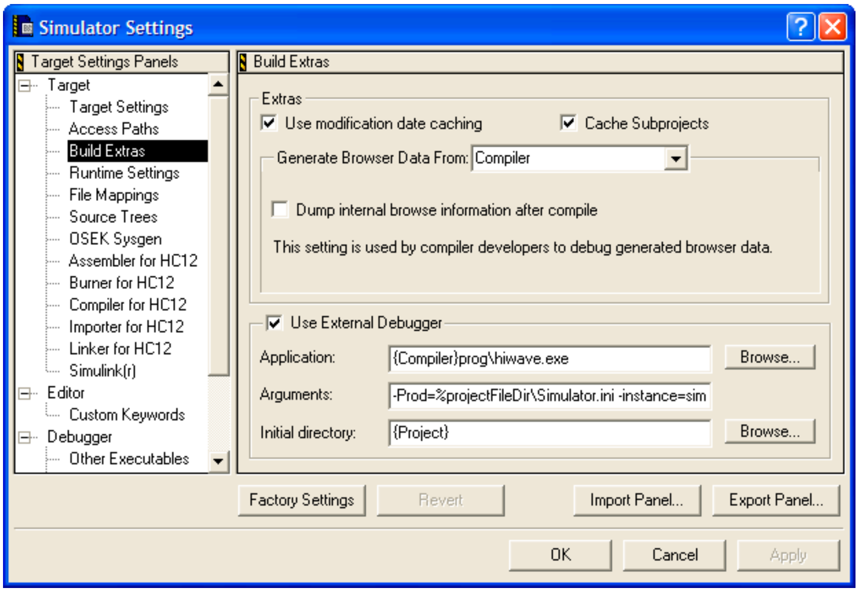
Depending on the CPU targets installed, you can choose from various linkers available in the linker drop box.

You can also select a libmaker. When a libmaker is set up, the build target is a library (`*.lib`) file. Furthermore, you may decide to rename the project's target by entering its name in the *Target Name:* text box.

Build Extras preference panel

Use the Build Extras Preference Panel (Figure 1.43) to get the compiler to generate browser information.

Figure 1.43 Build Extras preference panel



☑ Use modification date caching ☑ Cache Subprojects

Generate Browser Data From: Compiler

☐ Dump internal browse information after compile

This setting is used by compiler developers to debug generated browser data.

☑ Use External Debugger

Application: {Compiler}prog\hiwave.exe

Arguments: -Prod=%projectFileDir\Simulator.ini -instance=sim

Initial directory: {Project}

Enable the '*Use External Debugger*' check box to use the external simulator or debugger. Define the path to the debugger, which is either absolute (for example, 'C:\Freescale\prog\hiwave.exe'), or installation-relative (for example, '{Compiler}prog\hiwave.exe').

Additional command-line arguments passed to the debugger are specified in the Arguments box. In addition to the normal arguments (refer to your simulator or debugger documentation), the following `'% macros'` can also be specified:

- `%sourceFilePath`
- `%sourceFileDir`
- `%sourceFileName`
- `%sourceLineNumber`
- `%sourceSelection`
- `%sourceSelUpdate`
- `%projectFilePath`
- `%projectFileDir`
- `%projectFileName`
- `%projectSelectedFiles`

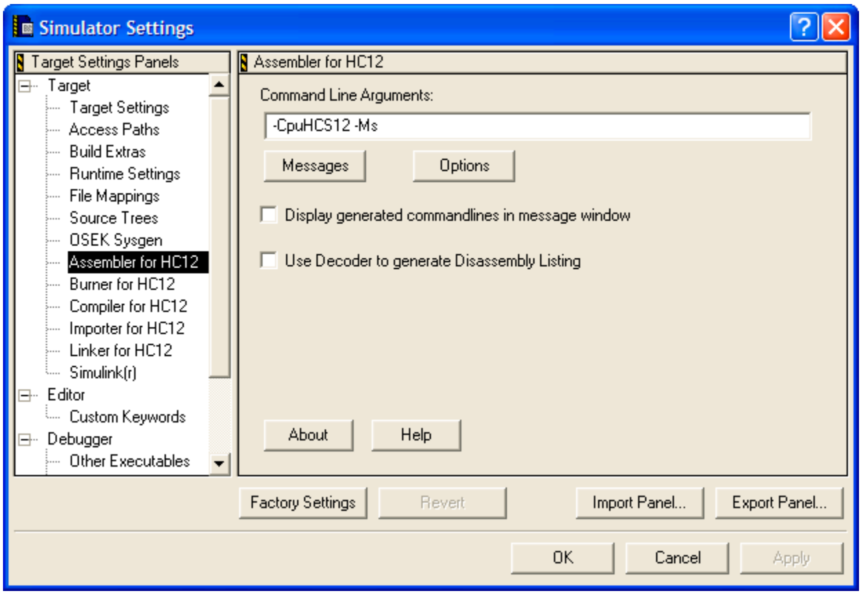
- `%targetFilePath`
- `%targetFileDir`
- `%targetFileName`
- `%currentTargetName`
- `%symFilePath`
- `%symFileDi`
- `%symFileName`

Assembler for HC12 preference panel

The Assembler for HC12 preference panel (Figure 1.44) contains the following:

- *Command Line Arguments*: Command-line options are displayed. You can add, delete, or modify the options by hand, or by using the *Messages* and *Options* buttons below.
  - Messages: Button to open the *Messages* dialog box
  - Options: Button to open the *Options* dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- *Use Decoder to generate Disassembly Listing*: The built-in listing file generator is used to produce the disassembly listing. If this check box is set, the external decoder is enabled.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

Figure 1.44 Assembler for HC12 preference panel



Assembler for HC12

Command Line Arguments: -CpuHCS12 -Ms

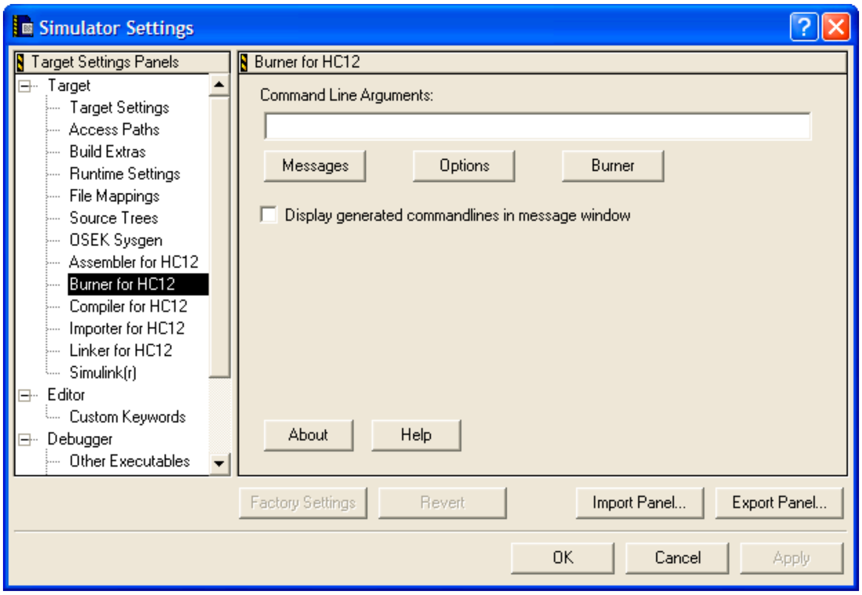
☐ Display generated commandlines in message window

☐ Use Decoder to generate Disassembly Listing

Burner preference panel

The Burner Plug-In Function: The `*.bbl` (batch burner language) files are mapped to the Burner Plug-In in the File Mappings Preference Panel. Whenever a `*.bbl` file is in the project file, the `*.bbl` file is processed during the post-link phase using the settings in the Burner Preference Panel (Figure 1.45).

Figure 1.45 Burner for HC12 preference panel



The Burner for HC12 preference panel contains the following:

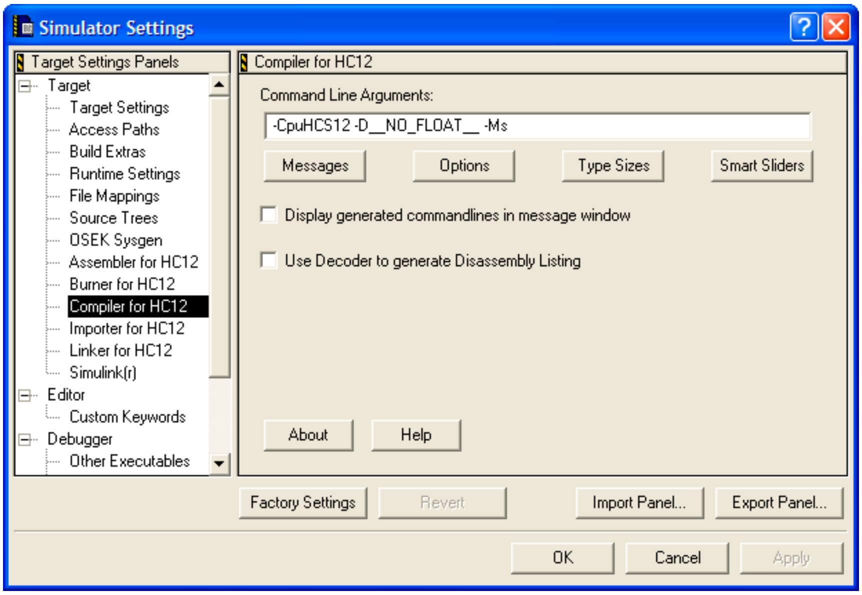
- *Command Line Arguments*: The actual command line options are displayed. You can add, delete, or modify the options manually, or use the Messages, Options, and Burner buttons listed below.
  - *Messages*: Opens the Messages dialog box
  - *Options*: Opens the Options dialog box
  - *Burner*: Opens the Burner dialog box
- `Display generated commandlines in message window`: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the 'Errors & Warnings' window. With this check box set, the complete command line is passed to the tool.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

Compiler for HC12 preference panel

The plug-in Compiler Preference Panel (Figure 1.46) contains the following:

- *Command Line Arguments*: Command line options are displayed. You can add, delete, or modify the options manually, or use the Messages, Options, Type Sizes, and Smart Sliders buttons listed below.
  – *Messages*: Opens the Messages dialog box
  – *Options*: Opens the Options dialog box
  – *Type Sizes*: Opens the Standard Type Size dialog box
  – *Smart Sliders*: Opens the Smart Slider dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the '*Errors & Warnings*' window. With this check box set, the complete command line is passed to the tool.
- *Use Decoder to generate Disassembly Listing*: Checking this check box enables the external decoder to generate a disassembly listing.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

Figure 1.46 Compiler for HC12 preference panel



Command Line Arguments: -CpuHCS12 -D__NO_FLOAT__ -Ms

☐ Display generated commandlines in message window

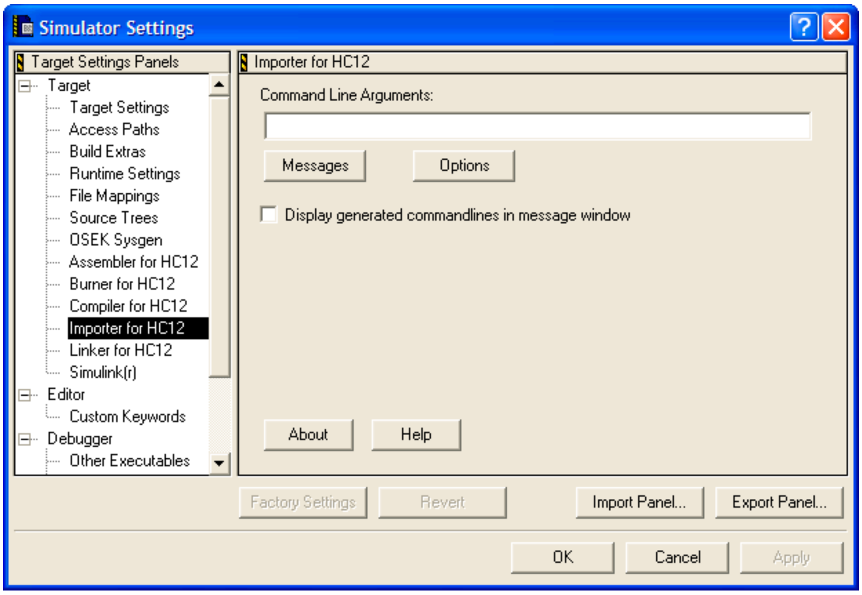
☐ Use Decoder to generate Disassembly Listing

Importer for HC12 preference panel

The plug-in Importer Preference Panel (Figure 1.47) contains the following controls:

- *Command-line Arguments*: Command-line options are displayed. You can add, delete, or modify the options manually, or use the Messages or Options buttons listed below.
  - *Messages*: Opens the Messages dialog box
  - *Options*: Opens the Options dialog box
- *Display generated commandlines in message window*: The plug-in filters the messages produced so that only Warning, Information, or Error messages are displayed in the 'Errors & Warnings' window. With this check box set, the complete command line is passed to the tool.
- *About*: Provides status and version information.
- *Help*: Opens the help file.

Figure 1.47 Importer preference panel

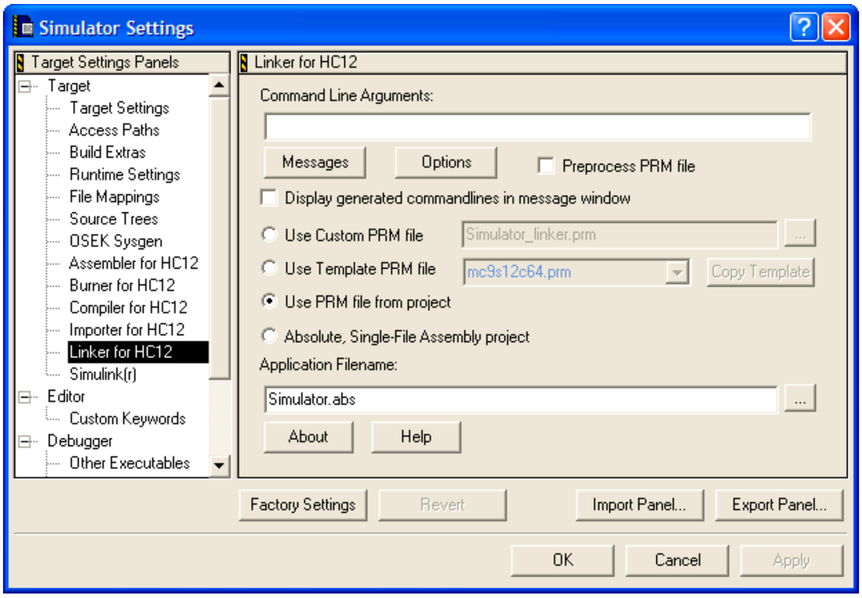


Linker for HC12 preference panel

This preference panel (Figure 1.48) displays in the *Target Settings Panel* if the Linker is selected. The plug-in preference panel contains the following controls:

- *Command-line Arguments*: Command-line options are displayed. You can add, delete, or modify the options manually, or use the *Messages* or *Options* buttons listed below.
  - *Messages*: Opens the *Messages* dialog box
  - *Options*: Opens the *Options* dialog box
- *Preprocess PRM file*: When checked, the preprocessor of the ANSI-C compiler is used to preprocess the PRM file prior to the linking step. In the PRM file, all ANSI-C preprocessor conditions like conditional inclusion (#if) are available. The same preprocessor macros as in ANSI-C code can be used (e.g., #ifdef __SMALL__).
- *Display generated commandlines in message window*: The plug-in filters the messages produced, so that only Warning, Information, or Error messages are displayed in the '*Errors & Warnings*' window. With this check box set, the complete command line is passed to the tool.
- *Use Custom PRM file*: Specifies a custom linker parameter file in the edit box. Use the browse button (...) to browse for a file.
- *Use Template PRM file*: With this radio control set, you can select one of the pre-made PRM files located in the templates directory (usually `C:\Freescale\templates\<target>\prm`). By employing the *'Copy Template'* button, the user can copy a template PRM file into the project to maintain a local copy.
- *Application Filename*: The output filename is specified.
- *About*: Provides status and version information.
- *Help*: Button to open the tool help file directly.

Figure 1.48 Linker preference panel



Command Line Arguments:

[ ] Preprocess PRM file

[ ] Display generated commandlines in message window

○ Use Custom PRM file: Simulator_linker.prm

○ Use Template PRM file: mc9s12c64.prm

● Use PRM file from project

○ Absolute, Single-File Assembly project

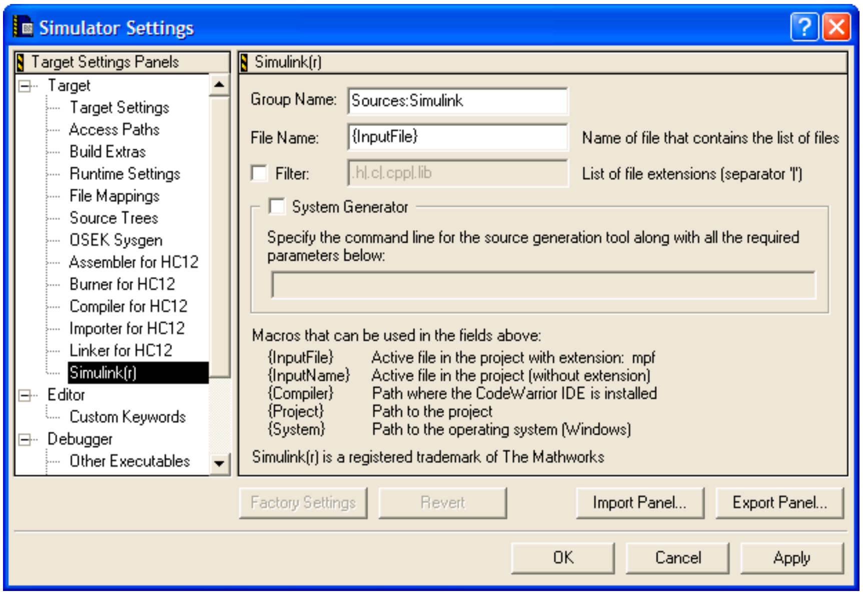
Application Filename: Simulator.abs

Simulink preference panel

The plug-in Importer Preference Panel (Figure 1.47) contains the following controls:

- *Group Name*
- *File Name*: Filename of the listing file.
- *Filter*: List of file extensions. Use “|” for separators.
- *System Generator*: Specify the command line of the source-generation tool.

Figure 1.49 Simulink preference panel



Group Name: Sources:Simulink

File Name: {InputFile} Name of file that contains the list of files

☐ Filter: .h|.c|.cpp|.lib List of file extensions (separator '|')

☐ System Generator

Specify the command line for the source generation tool along with all the required parameters below:

Macros that can be used in the fields above:

{InputFile}	Active file in the project with extension: mpf
{InputName}	Active file in the project (without extension)
{Compiler}	Path where the CodeWarrior IDE is installed
{Project}	Path to the project
{System}	Path to the operating system (Windows)

Simulink(r) is a registered trademark of The Mathworks

CodeWarrior Tips and Tricks

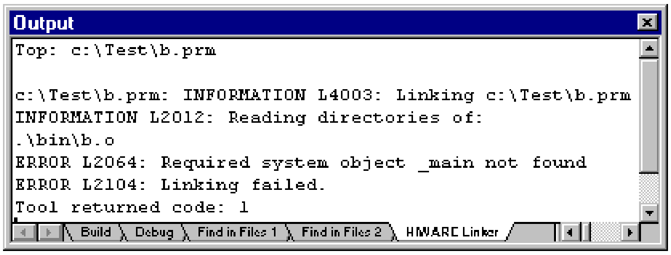
If the Simulator or Debugger cannot be launched, check the settings in the *Build Extras Preference Panel*.

If the data folder of the project is deleted, then some project-related settings may also have been deleted.

If a file cannot be added to the project, its file extension may be absent from the *File Mappings Preference Panel*. Adding this file's extension to the listing in the *File Mappings Preference Panel* should correct this.

If it is suspected that project data is corrupted, export and re-import the project using *File -> Export Project and File -> Import Project*.

Figure 1.50 Compiler Log Display



```
Top: c:\Test\b.prm

c:\Test\b.prm: INFORMATION L4003: Linking c:\Test\b.prm
INFORMATION L2012: Reading directories of:
.\bin\b.o
ERROR L2064: Required system object _main not found
ERROR L2104: Linking failed.
Tool returned code: 1
```

Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)

Use the procedure in Listing 1.5 to integrate the Tools into the Microsoft Visual Studio (Visual C++).

Listing 1.5 Integration as Additional Tools

(1) Start Visual Studio.

(2) Select the menu *Tools->Customize*.

(3) Select the *Tools* Tab.

(4) Add a new tool using the *New* button, or by double-clicking
on the last empty entry in the *Menu contents* list.

(5) Type in the name of the tool to display in the menu (for example, *Linker*).

(6) In the *Command* field, type in the name and path of the piper tool (for example, ‘C:\Freescale\prog\piper.exe’.

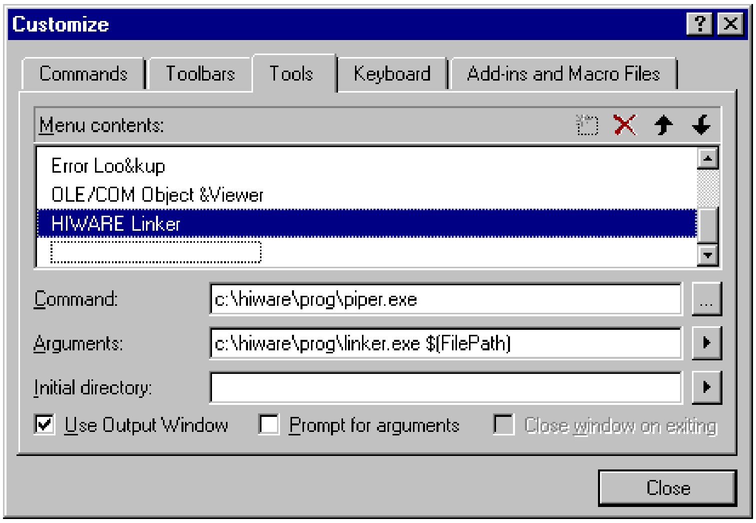
(7) In the *Arguments* field, type in the name of the tool to be started with any command line options (for example, `-N`) and the $(`FilePath`) Visual variable
(for example, ‘`C:\Freescale\prog\linker.exe -N $(FilePath)`’).

(8) Check *Use Output Window*.

(9) Uncheck *Prompt for arguments*.

Proceed as above for all other tools (for example, compiler, decoder).
Close the *‘Customize’* dialog box (Figure 1.51).

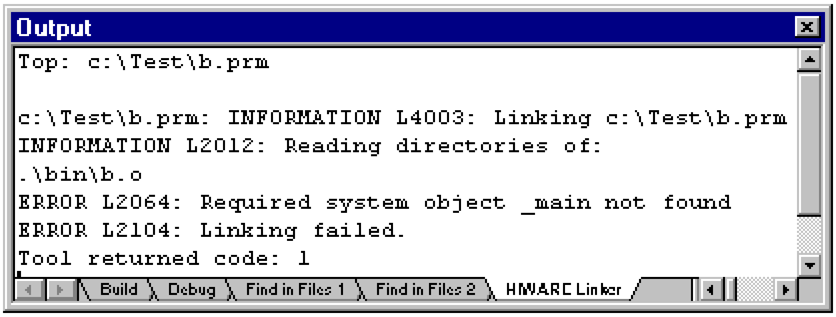
Figure 1.51 Customize dialog box



Command: c:\hiware\prog\piper.exe
Arguments: c:\hiware\prog\linker.exe $(FilePath)
Initial directory:
[✓] Use Output Window
[ ] Prompt for arguments
[ ] Close window on exiting

This allows the active file to be compiled or linked in the Visual Editor (`'$(FilePath)`). Tool messages are reported in a separate *Visual* output window (Figure 1.52). Double click on the output entries to jump to the right message position (message feedback).

Figure 1.52



```
Top: c:\Test\b.prm

c:\Test\b.prm: INFORMATION L4003: Linking c:\Test\b.prm
INFORMATION L2012: Reading directories of:
.\bin\b.o
ERROR L2064: Required system object _main not found
ERROR L2104: Linking failed.
Tool returned code: 1
```

Use the procedure in Listing 1.6 to integrate the Toolbar in Microsoft Visual Studio (Visual C++).

Listing 1.6 Integration with Visual Studio Toolbar

(1) Start Visual Studio.
Make sure that all tools are integrated as *Additional Tools*.

(3) Select the menu *Tools* -> *Customize*.

(4) Select the *Toolbars* Tab.

(5) Select *New* and enter a name (for example, `CodeWarrior Build Tools`). A new empty toolbar named *CodeWarrior Build Tools* appears on your screen.

(6) Select the *Commands* Tab.

(7) In the *Category* drop down box, select *Tools*.
On the right side many 'hammer' tool images appear, each with a number. The number corresponds to the entry in the *Tool* menu. Usually the first user-defined tool is tool number 7. (The Linker was set up in *Additional Tools* above.)

(8) Drag the selected tool image to the *CodeWarrior Build Tools* toolbar.

All default tool images look the same, making it difficult to know which tool has been launched. You should associate a name with them.
(a) Right-click on a tool in the *CodeWarrior Build Tools* to open the context menu of the button.
(b) Select *Button Appearance...* in the context menu.
(c) Select *Image and Text*.
(d) Enter the tool name to associate with the image in *Button text:* (for example, *Linker*).

(9) Repeat the above for all the desired tools to appear in the toolbar.

(10) Close the *Customize* dialog box after all the Build Tools are entered into the Toolbar.

This enables the tools to be started from the toolbar.

The Compiler provides the following language settings:

- ANSI-C: The compiler can behave as an ANSI-C compiler. It is possible to force the compiler into a strict ANSI-C compliant mode.
- language extensions that are specially designed for more efficient embedded systems programming.

Object-File Formats

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

NOTE Be careful and do not mix object-file formats. *Both the HIWARE and the ELF/DWARF object files use the same filename extensions*.

HIWARE Object-File Format

The HIWARE Object-File Format is a vendor-specific object-file format defined by HIWARE AG. This object-file format is very compact. The object file sizes are smaller than the ELF/DWARF object files. This smaller size enables faster file operations on the object files. The object-file format is also easy to support by other tool vendors. The object-file format supports ANSI-C and Modula-2.

Each other tool vendor must support this object-file format explicitly. Note that there is also a lack of extensibility, amount of debug information, and C++ support. For example, using the full flexibility of the Compiler Type Management is not supported in the HIWARE Object-file Format.

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

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.

Object-file sizes are large compared with the HIWARE object-file format.

NOTE ANSI-C and Modula-2 are supported in this object-file format.

Tools

The CodeWarrior Development Studio contains the following Tools, among others:

Compiler

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

Note that not all Compiler backends support both ELF/DWARF and the HIWARE Object-File formats. Some only support one of the two.

Decoder

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

Linker

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

Simulator or Debugger

The Simulator or Debugger supports both object-file formats.

Mixing Object-File Formats

Mixing HIWARE and ELF object files is not possible. 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.

2 Graphical User Interface

The Graphical User Interface (GUI) tool provides both a simple and a powerful user interface:

- Graphical User Interface
- Command-Line User Interface
- Online Help
- Error Feedback
- Easy integration into other tools (for example, CodeWarrior, CodeWright, MS Visual Studio, WinEdit, ...)

This chapter describes the user interface and provides useful hints. Its major elements are:

- “Launching the Compiler” on page 89
- “Tip of the Day” on page 91
- “Main Window” on page 92
- “Window Title” on page 92
- “Content Area” on page 92
- “Toolbar” on page 93
- “Status Bar” on page 94
- “Menu Bar” on page 94
- “Standard Types dialog box” on page 109
- “Option Settings dialog box” on page 110
- “Compiler Smart Control dialog box” on page 112
- “Message Settings dialog box” on page 113
- “About ... dialog box” on page 116
- “Specifying the Input File” on page 117

Launching the Compiler

Start the compiler using:

- The Windows Explorer

- An Icon on the desktop
- An Icon in a program group
- Batch and command files
- Other tools (Editor, Visual Studio, etc.)

Interactive Mode

If the compiler is started with no input (that means no options and no input files), then the graphical user interface (GUI) is active (interactive mode). This is usually the case if the compiler is started using the Explorer or using an Icon.

Batch Mode

If the compiler is started with arguments (options and/or input files), then it is started in batch mode (Listing 2.1).

Listing 2.1 Specify the line associated with an icon on the desktop.

```
C:\Freescale\prog\chc12.exe -F2 a.c d.c
```

In batch mode, the compiler does not open a window. It is displayed in the taskbar only for the time it processes the input and terminates afterwards (Listing 2.2).

Listing 2.2 Commands are entered to run as shown below.

```
C:\> C:\Program Files\Freescale\CW for HC12 V4.5\prog\chc12.exe -F2
a.c d.c
```

Message output (stdout) of the compiler is redirected using the normal redirection operators (for example, `'>'` to write the message output to a file), as shown in Listing 2.3:

Listing 2.3 Command-line message output is redirected to a file.

```
C:\> C:\Program Files\Freescale\CW for HC12 V4.5\prog\chc12.exe -F2
a.c d.c > myoutput.o
```

The command line process returns after starting the compiling process. It does not wait until the started process has terminated. To start a process and wait for termination (for example, for synchronization), use the `'start'` command under Windows NT/95/98/Me/2000/XP, or use the `'/wait'` options (see windows help `'help start'`). Using *'start /wait'* (Listing 2.4) you can write perfect batch files (for example, to process your files).

Listing 2.4 Start a compilation process and wait for termination.

```
C:\> start /wait C:\Freescale\prog\chc12.exe -F2 a.c d.c
```

Tip of the Day

When the application is started, a standard *Tip of the Day* (Figure 2.1) window is opened containing the last news and tips.

The *Next Tip* button displays the next tip about the application.

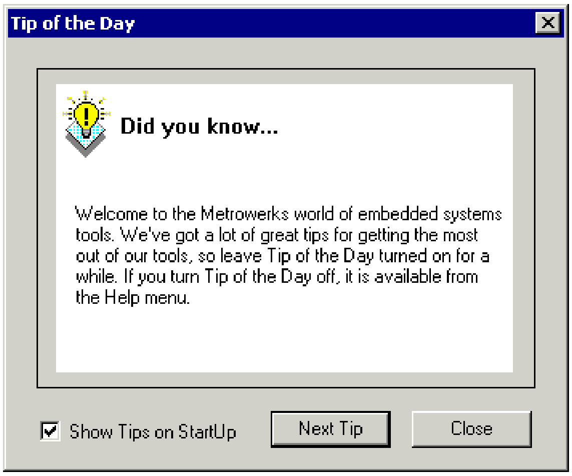
If it is not desired for the *Tip of the Day* window to open automatically when the application is started, uncheck the check box *Show Tips on StartUp*.

NOTE This configuration entry is stored in the local project file.

To enable automatic display from the standard *Tip of the Day* window when the application is started, select the entry *Help* | *Tip of the Day....* The *Tip of the Day* window opens. Check the box *Show Tips on StartUp*.

Click *Close* to close the *Tip of the Day* window.

Figure 2.1 Tip of the Day Dialog

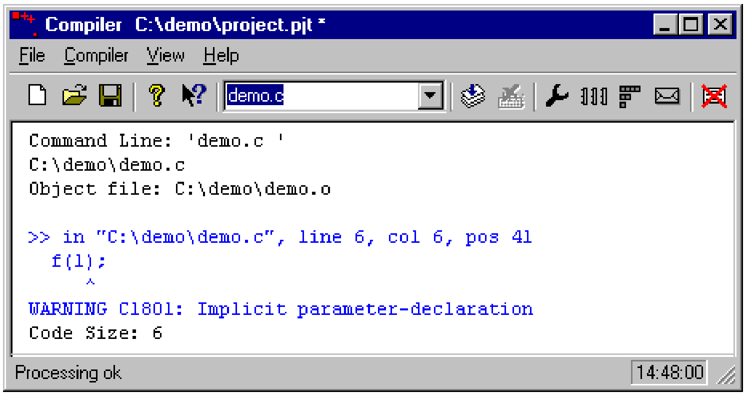


Welcome to the Metrowerks world of embedded systems tools. We've got a lot of great tips for getting the most out of our tools, so leave Tip of the Day turned on for a while. If you turn Tip of the Day off, it is available from the Help menu.

Main Window

This Main Window (Figure 2.2) is only visible on the screen when a filename is not specified while starting the application. The application window provides a window title, a menu bar, a toolbar, a content area, and a status bar.

Figure 2.2 Main Window



```
Command Line: 'demo.c '
C:\demo\demo.c
Object file: C:\demo\demo.o

>> in "C:\demo\demo.c", line 6, col 6, pos 41
   f(1);
      ^
WARNING C1801: Implicit parameter-declaration
Code Size: 6
```

Window Title

The window title displays the application name and the project name. If there is no project currently loaded, "*Default Configuration*" is displayed. An asterisk (`*`) after the configuration name is present if any value has changed but has not yet been saved.

NOTE Changes to options, the Editor Configuration, and the application appearance can make the `"*"` appear.

Content Area

The content area is used as a text container, where logging information about the process session is displayed. This logging information consists of:

- The name of the file being processed
- The whole names (including full path specifications) of the files processed (main C file and all files included)
- An error, warning, and information message list

- The size of the code generated during the process session

When a file is dropped into the application window content area, the corresponding file is either loaded as configuration data, or processed. It is loaded as configuration data if the file has the "`*.ini`" extension. If the file does not contain this extension, the file is processed with the current option settings.

All text in the application window content area can contain context information. The context information consists of two items:

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

File context information is available for all output where a text file is considered. It is also available for all source and include files, and for messages which do concern a specific 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. The right mouse button can also be used to open a context menu. The context menu contains an **"***Open ...*" entry if a file context is available. If a file cannot be opened although a context menu entry is present, refer to 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.

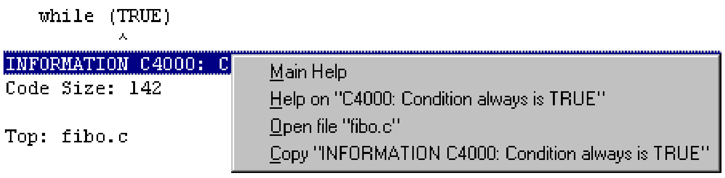
- Select one line of the message and *press F1*.

  If the selected line does not have a message number, the main help is displayed.
- Press *Shift-F1* and then click on the message text.

  If the point clicked at does not have a message number, the main help is displayed.
- Click with the right mouse at the message text and select *"Help on ..."*.

  This entry is available only if a message number is available (Figure 2.3).

Figure 2.3 Online Help Dialog



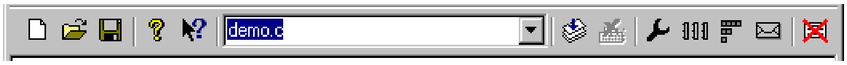
```
while (TRUE)
       ^
INFORMATION C4000: C
Code Size: 142

Top: fibo.c
```

Main Help
Help on "C4000: Condition always is TRUE"
Open file "fibo.c"
Copy "INFORMATION C4000: Condition always is TRUE"

Toolbar

The three buttons on the left in the Toolbar (Figure 2.4) are linked with the corresponding entries of the *File* menu. The next button opens the *About...* dialog box. After pressing the context help button (or the shortcut *Shift F1*), the mouse cursor changes its form and displays a question mark beside the arrow. The help file is called for the next item which is clicked. It is clicked on menus, toolbar buttons, and on the window area to get help specific for the selected topic.

Figure 2.4 Toolbar



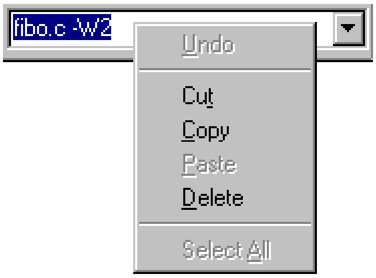
The command line history contains a list of the commands executed. Once a command is selected or entered in history, clicking *Compile* starts the execution of the command. Use the F2 keyboard shortcut key to jump directly to the command line. In addition, there is a context menu associated with the command line (Figure 2.5):

The *Stop* button stops the current process session.

The next four buttons open the option setting, the smart slider, type setting, and the message setting dialog box.

The last button clears the content area (Output Window).

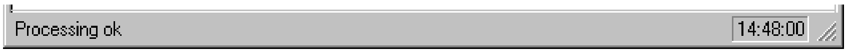
Figure 2.5 Command line Context Menu



Status Bar

When pointing to a button in the toolbar or a menu entry, the message area displays the function of the button or menu entry being pointed to.

Figure 2.6 Status Bar



Menu Bar

Table 2.1 lists and describes the menus available in the menu bar (Figure 2.7):

Table 2.1 Menus in the Menu Bar

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

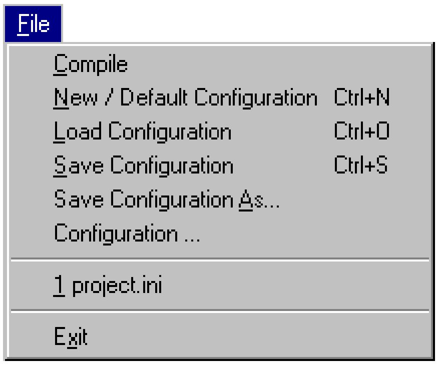
Figure 2.7 Menu Bar

File Menu

Save or load configuration files from the File Menu (Figure 2.8). A configuration file contains the following information:

- The application option settings specified in the application dialog boxes
- The Message Settings that specify which messages to display and which messages to treat as error messages
- The list of the last command line executed and the current command line being executed
- The window position
- The Tips of the Day settings, including if enabled at startup and which is the current entry

Figure 2.8 File Menu


File
Compile
New / Default Configuration Ctrl+N
Load Configuration Ctrl+O
Save Configuration Ctrl+S
Save Configuration As...
Configuration ...
1 project.ini
Exit

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.

Table 2.2 describes all the commands that are available from the File Menu:

Table 2.2 File Menu Commands

Menu entry	Description
Compile	Opens a standard Open File box. The configuration data stored in the selected file is loaded and used by a future session.
New / Default Configuration	Resets the application option settings to the default value. The application options which are activated per default are specified in section *Command Line Options* in this document
Load Configuration	Opens a standard Open File box. The configuration data stored in the selected file is loaded and used by a future session.
Save Configuration	Saves the current settings.
Save Configuration As...	Opens a standard Save As box. The current settings are saved in a configuration file which has the specified name. See Local Configuration File (usually project.ini).
Configuration...	Opens the *Configuration* dialog box to specify the editor used for error feedback and which parts to save with a configuration.

Table 2.2 File Menu Commands (*continued*)

Menu entry	Description
1. .... project.ini 2.	Recent project list. This list is accessed to open a recently opened project again.
Exit	Closes the application.

Editor Settings dialog box

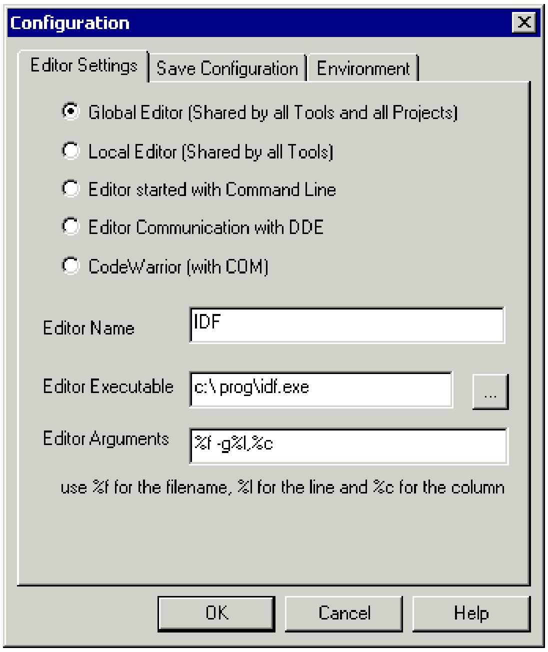
The Editor Settings dialog box has a main selection entry. Depending on the main type of editor selected, the content below changes.

These main Editor Setting entries are described on the following pages.

Global Editor configuration

The Global Editor (Figure 2.9) 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.

Figure 2.9 Global Editor configuration



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

Editor Name: IDF

Editor Executable: c:\ prog\idf.exe

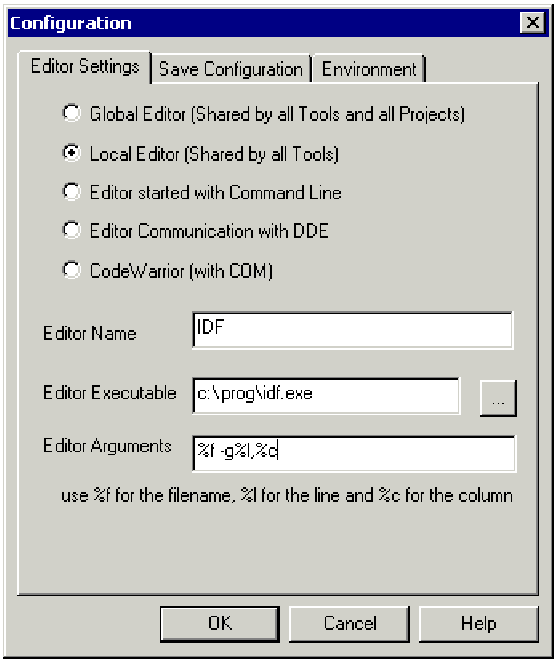
Editor Arguments: %f -g%l,%c

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

Local Editor configuration

The Local Editor (Figure 2.10) 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.

Figure 2.10 Local Editor configuration



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

Editor Name: IDF
Editor Executable: c:\prog\idf.exe
Editor Arguments: %f -g%l,%c

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

Editor started with Command Line

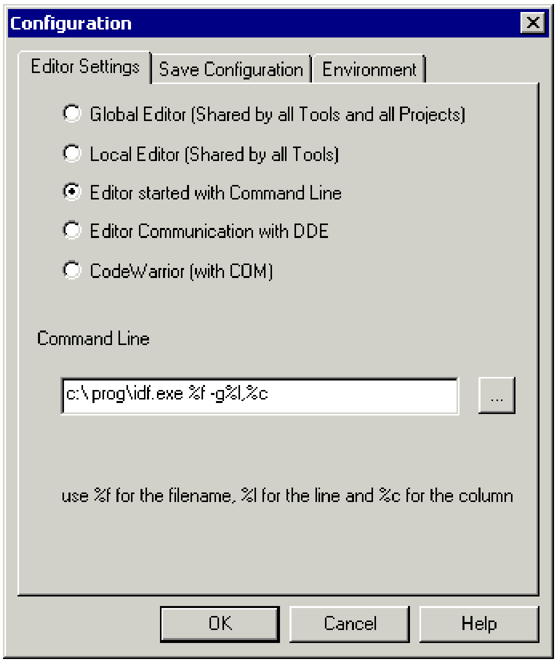
When this editor type (Figure 2.11) is selected, a separate editor is associated with the application for error feedback. The configured editor is not used for error feedback.

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. (See the Modifiers section below.)

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

Figure 2.11 Editor Started with 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)

Command Line

`c:\prog\idf.exe %f -g%l,%c`

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

Examples:

For CodeWright V6.0 version, use (with an adapted path to the `cw32.exe` file):

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

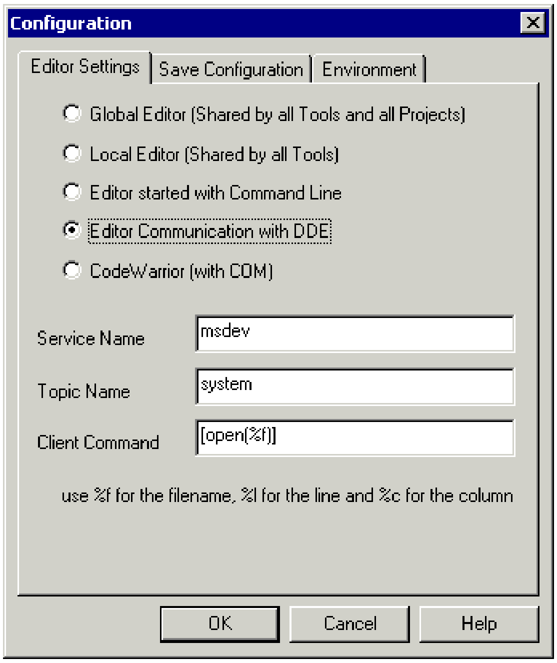
For the WinEdit 32-bit version, use (with an adapted path to the winedit.exe file):

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

Editor Started with DDE

Enter the service and topic names and the client command for the DDE connection to the editor (Microsoft Developer Studio - Figure 2.12 or UltraEdit-32 - Figure 2.13). The entries for Topic Name and Client Command can have modifiers for the filename, line number, and column number as explained in “Modifiers” on page 103.

Figure 2.12 Editor Started with DDE (Microsoft Developer Studio)



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

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

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

For Microsoft Developer Studio, use the settings in Listing 2.5.

Listing 2.5 .Microsoft Developer Studio configuration

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

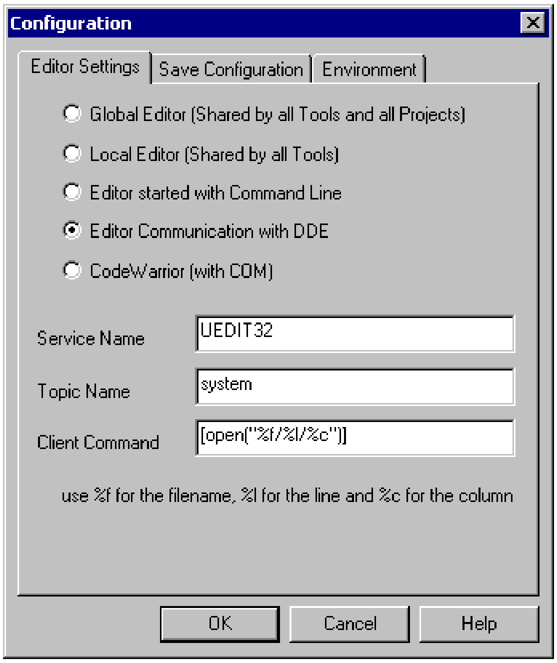
UltraEdit-32 is a powerful shareware editor. It is available from www.idmcomp.com or www.ultraedit.com, email idm@idmcomp.com. For UltraEdit, use the following settings (Listing 2.6).

Listing 2.6 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) has to be started or otherwise the DDE communication will fail.

Figure 2.13 Editor Started with DDE (UltraEdit-32)



( ) 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)

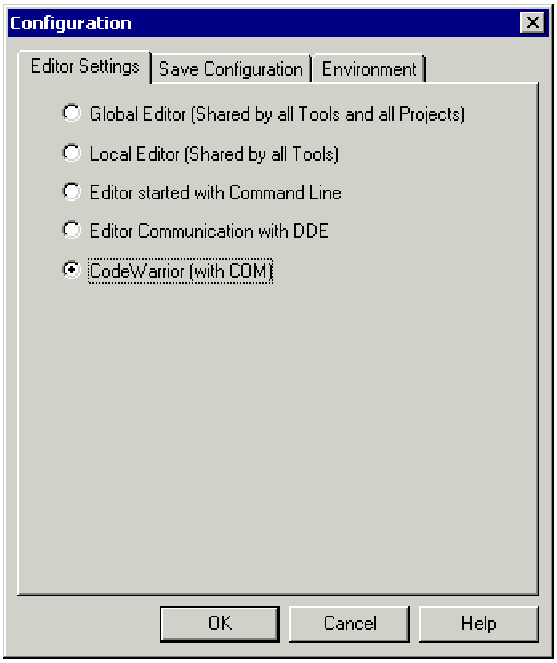
Service Name	UEDIT32
Topic Name	system
Client Command	[open("%f/%l/%c")]

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

CodeWarrior (with COM)

If CodeWarrior with COM (Figure 2.14) is enabled, the CodeWarrior IDE (registered as COM server by the installation script) is used as the editor.

Figure 2.14 CodeWarrior (with COM)



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

Modifiers

The 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`
Please check the editor manual to define the command line which should be used to start the editor.

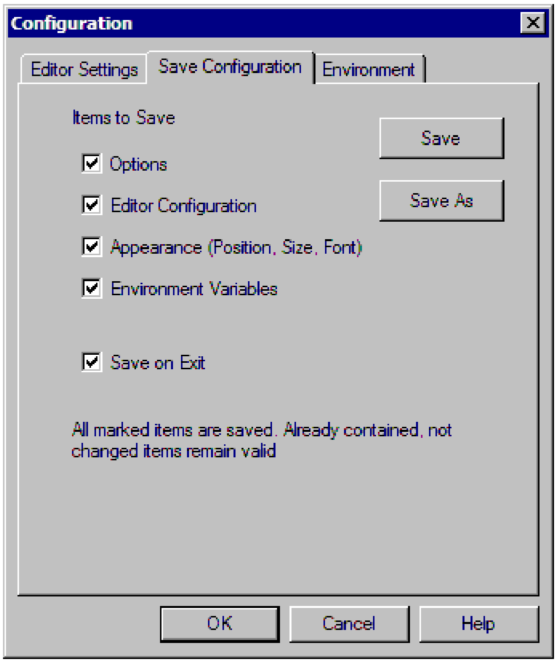
Save Configuration dialog box

All save options are located on the second page of the configuration dialog box.

Use the Save Configuration dialog box to configure which parts of your configuration are stored into a project file.

This Save Configuration dialog box (Figure 2.15) offers the following options:

Figure 2.15 Save Configuration dialog box



Items to Save

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

All marked items are saved. Already contained, not changed items remain valid

- Options

  The current option and message setting is saved when a configuration is written. By disabling this option, the last saved content remains valid.

- Editor Configuration

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

- Appearance

  This saves topics consisting of many parts such as the window position (only loaded at startup time) and the command line content and history. These settings are saved when a configuration is written.

- Environment Variables

  The environment variable changes done in the Environment property sheet are saved.

NOTE By disabling selective options only some parts of a configuration file are written. For example, when the best options are found, the save option mark is removed. 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.
- 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.

Environment Configuration Dialog Box

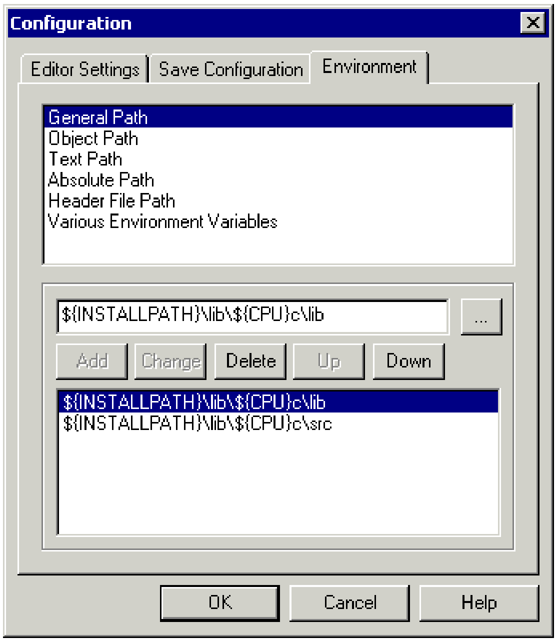
This Environment Configuration dialog box (Figure 2.16) is used to configure the environment. The content of the dialog box is read from the actual project file out of the section [Environment Variables].

The following environment variables are available (Listing 2.1):

Listing 2.7 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 2.16 Environment Configuration dialog box



```
${INSTALLPATH}\lib\${CPU}c\lib
```

```
${INSTALLPATH}\lib\${CPU}c\lib
${INSTALLPATH}\lib\${CPU}c\src
```

The following buttons are available on this dialog box (Table 2.3):

Table 2.3 Functions of the buttons on the Environment Configuration dialog box

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

The variables are written to the project file only if the *Save* button is pressed (or use *File->Save Configuration*, or *CTRL-S*). In addition, the environment is specified if it is to be written to the project in the *Save Configuration* dialog box.

Compiler Menu

This menu (Figure 2.17) enables the application to be customized. Application options are graphically set as well as defining the optimization level.

Figure 2.17 Compiler Menu

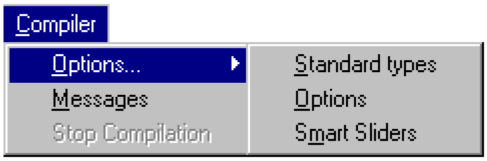


Table 2.4 Compiler Menu options

Menu entry	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. (See “Standard Types dialog box” on page 109.)
Advanced	Allows you to define the options which must be activated when processing an input file. (See “Option Settings dialog box” on page 110.).
Smart Slider	Allows you to define the optimization level you want to reach when processing the input file. (See “Compiler Smart Control dialog box” on page 112.)
Messages	Opens a dialog box, where the different error, warning, or information messages are mapped to another message class. (See “Message Settings dialog box” on page 113.)
Stop Compilation	Immediately stops the current processing session.

View Menu

The View Menu (Figure 2.18) 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. Table 2.5 lists the View Menu options.

Figure 2.18 View Menu

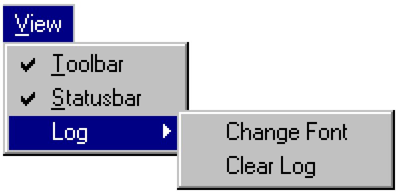


Table 2.5 View Menu options

Menu entry	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.

Help Menu

The Help Menu (Figure 2.19) enables you to either display or not display the Tip of the Day dialog box application startup. In addition, it provides a standard Windows Help entry and an entry to an About box. Table 2.6 defines the Help Menu options:

Figure 2.19 Help Menu

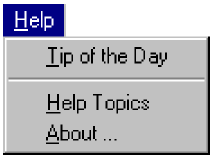


Table 2.6 Help Menu Options

Menu entry	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.

Standard Types dialog box

The Standard Types dialog box (Figure 2.20) 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 illegal to specify all types to a size of 32 bits. There is no type for 8 bits and 16 bits available for the Compiler. Note that if the HIWARE object-file Format is used instead of the ELF/DWARF object-file Format, the HIWARE Format does not support a size greater than 1 for the `char` type.

The following rules (Listing 2.8) apply when you modify the size associated with an ANSI-C standard type:

Listing 2.8 Size relationships for the ANSI-C standard types.

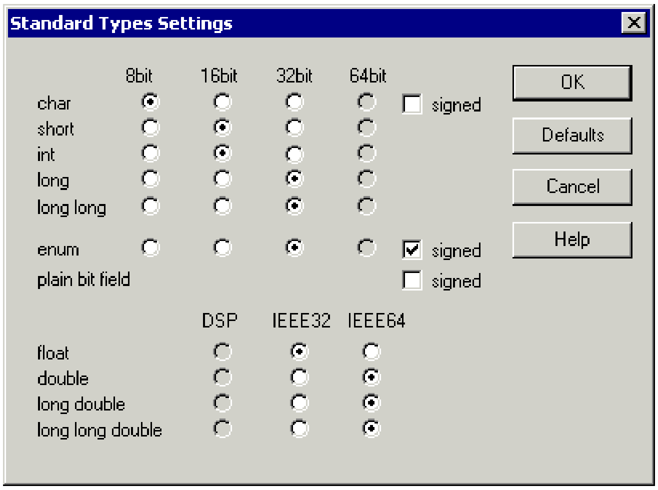
```
sizeof(char)  <= sizeof(short)
sizeof(short) <= sizeof(int)
sizeof(int)   <= sizeof(long)
sizeof(long)  <= sizeof(long long)
sizeof(float) <= sizeof(double)
sizeof(double)<= sizeof(long double)
```

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.

Figure 2.20 Standard Types Dialog Box



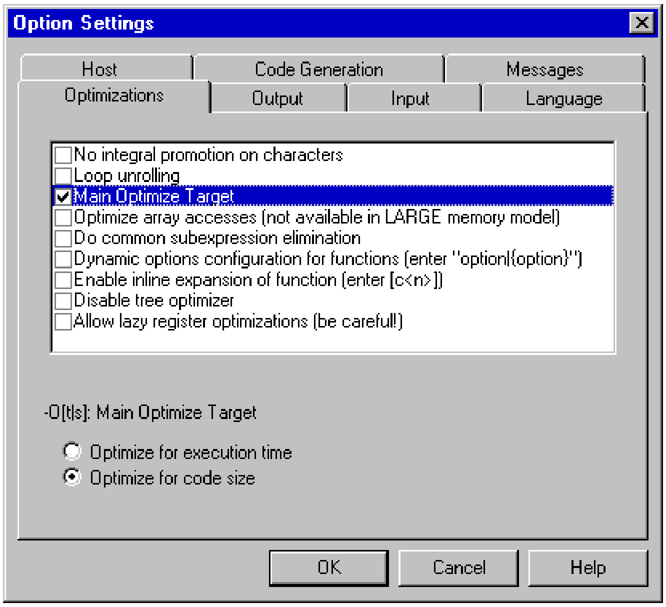
	8bit	16bit	32bit	64bit	
char	●	○	○	○	☐ signed
short	○	●	○	○	
int	○	●	○	○	
long	○	○	●	○	
long long	○	○	●	○	
enum	○	○	●	○	☑ signed
plain bit field					☐ signed

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

Option Settings dialog box

The Option Settings dialog box (Figure 2.21) 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). Table 2.7 lists the Option Settings dialog box selections.

Figure 2.21 Option Settings dialog box



[ ] No integral promotion on characters
[ ] Loop unrolling
[✓] Main Optimize Target
[ ] Optimize array accesses (not available in LARGE memory model)
[ ] Do common subexpression elimination
[ ] Dynamic options configuration for functions (enter "option|{option}")
[ ] Enable inline expansion of function (enter [c<n>])
[ ] Disable tree optimizer
[ ] Allow lazy register optimizations (be careful!)

-O[t|s]: Main Optimize Target

Optimize for execution time [✓ Optimize for code size]

Table 2.7 Option Settings dialog box selections

Group	Description
Optimizations	Lists optimization options.
Output	Lists options related to the output files generation (which kind of file should be generated).
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 (memory models, float format, ...).
Messages	Lists options controlling the generation of error messages.
Various	Lists options not related to the above options.

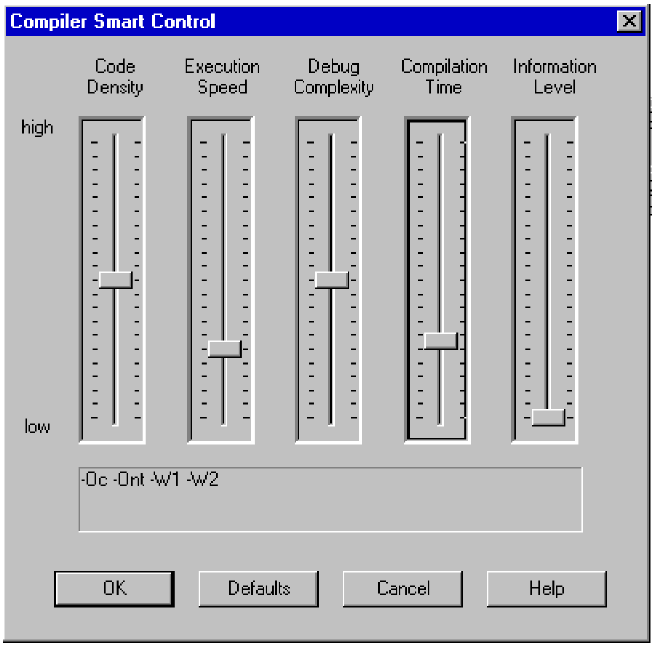
An application option is set when its check box is checked. To obtain a more detailed explanation about a specific option, select the option and press the F1 key or the help button. To select an option, click once on the option text. The option text is then displayed color-inverted. When the dialog box is opened and no option is selected, pressing the F1 key or the help button shows the help for this dialog box.

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.

Compiler Smart Control dialog box

The Compiler Smart Control Dialog Box (Figure 2.22) 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. See Table 2.8.

Figure 2.22 Compiler Smart Control dialog box



-Oc -Ont -W1 -W2

Table 2.8 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.
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.

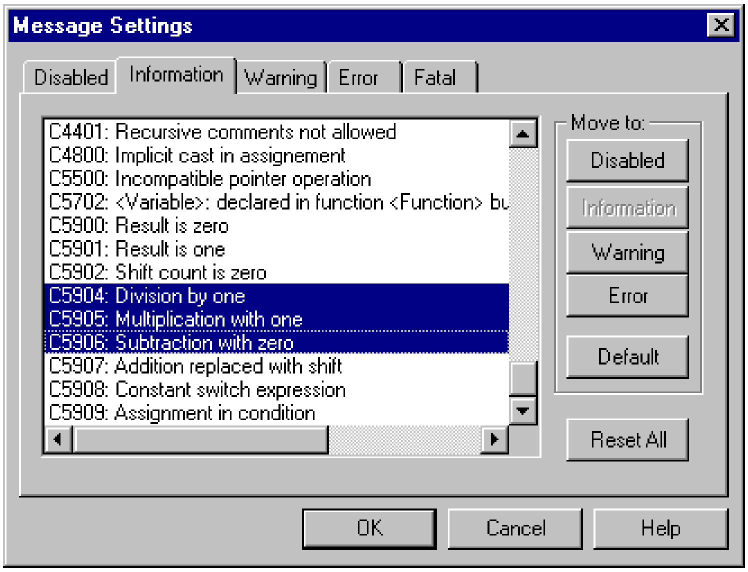
The command line is automatically updated with the options set in accordance with the settings of the different sliders.

Message Settings dialog box

The Message Settings dialog box (Figure 2.23) 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.) Table 2.9 lists and describes the buttons available in this dialog box.

Figure 2.23 Message Settings dialog box



C4401: Recursive comments not allowed
C4800: Implicit cast in assignement
C5500: Incompatible pointer operation
C5702: <Variable>: declared in function <Function> bu
C5900: Result is zero
C5901: Result is one
C5902: Shift count is zero
C5904: Division by one
C5905: Multiplication with one
C5906: Subtraction with zero
C5907: Addition replaced with shift
C5908: Constant switch expression
C5909: Assignment in condition

Table 2.9 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:.Table 2.10 lists the definitions for the message groups.

Table 2.10 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.
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.

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 (Listing 2.9).

Listing 2.9 Defining a warning message as an error message

(1) Click the *Warning* panel to display the list of all warning messages in the list box.

(2) 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.

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.

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.

Specifying the 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.

Use the Command Line in the Toolbar to Compile

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

Compiling a 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.

Compiling a file which has already been compiled

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.

Use the Entry File -> Compile

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.

Use 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 to compile it.

Message/Error Feedback

There are several ways to check where different errors or warnings have been detected after compilation. Listing 2.10 lists the format of the error messages and Listing 2.11 is a typical example of an error message.

Listing 2.10 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>
```

Listing 2.11 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.

Use Information from the 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.

Use a 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.

3

Environment

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

The major sections in this chapter are:

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 3.1):

Parameter: `<KeyName>"="<ParamDef>` (no spaces).

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

Listing 3.1 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.

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 DEFAULTDIR: Default Current Directory environment variable.

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

Environment Macros

You can use macros in your environment settings (Listing 3.2).

Listing 3.2 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\'`.

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 (e.g., `C:\WINDOWS`).

Listing 3.3 Typical Global Initialization File Locations

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

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

If the tool is started outside the `D:\INSTALL\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.

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.

Paths

A path list is a list of directory names separated by semicolons. Path names are declared using the following EBNF syntax (Listing 3.4).

Listing 3.4 EBNF path syntax

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

Most environment variables contain path lists directing where to look for files (Listing 3.5).

Listing 3.5 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 3.6 Setting an environment variable using recursive searching

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

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

Line Continuation

It is possible to specify an environment variable in an environment file (default.env) over different lines using the line continuation character '\' (see Listing 3.7).

Listing 3.7 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
```

will result in:

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

To avoid such problems, use a semicolon `';'` at the end of a path if there is a `'\'` at the end (Listing 3.8):

Listing 3.8 Using a semicolon to allow a multiline environment variable

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

Environment Variable Details

The remainder of this chapter describes each of the possible environment variables. Table 3.1 lists these description topics in their order of appearance for each environment variable.

Table 3.1 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.

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. It is used to globally specify options that should always be set. 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 3.9 Setting default values for environment variables (not recommended)

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

See also

Compiler Options

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

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)

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

Example

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

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.

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

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

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

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 will issue 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

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
- -Lm: List of Included Files in Make Format
- -Lo: Object File List

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

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

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

4

Files

This chapter describes input and output files and file processing.

- “Input Files” on page 141
- “Output Files” on page 142
- “File Processing” on page 143

Input Files

The following input files are described:

- Source Files
- Include Files

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.

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.

Output Files

The following output files are described:

- Object Files
- Error Listing

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.

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 environment variable).

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' *Find Next Error* command to display both the error positions and the error messages.

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.

Batch Mode (Compiler Window not Open)

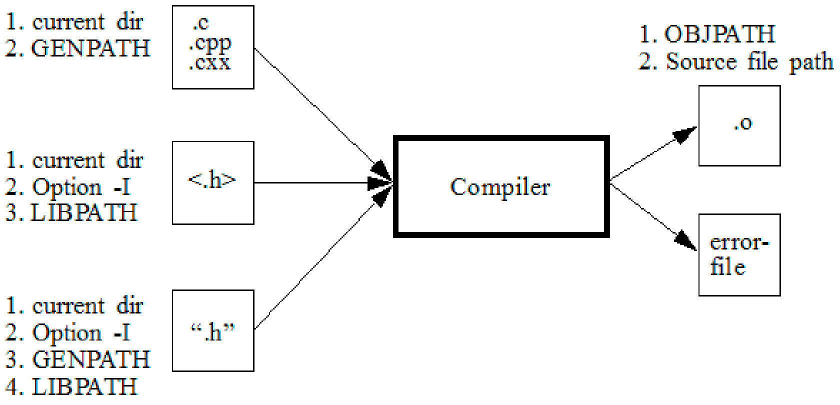
If ERRORFILE is set, the Compiler creates a message file named as specified in this environment variable.

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

File Processing

Figure 4.1 shows how file processing occurs with the Compiler:

Figure 4.1 Files used with the Compiler



5 Compiler Options

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.

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.

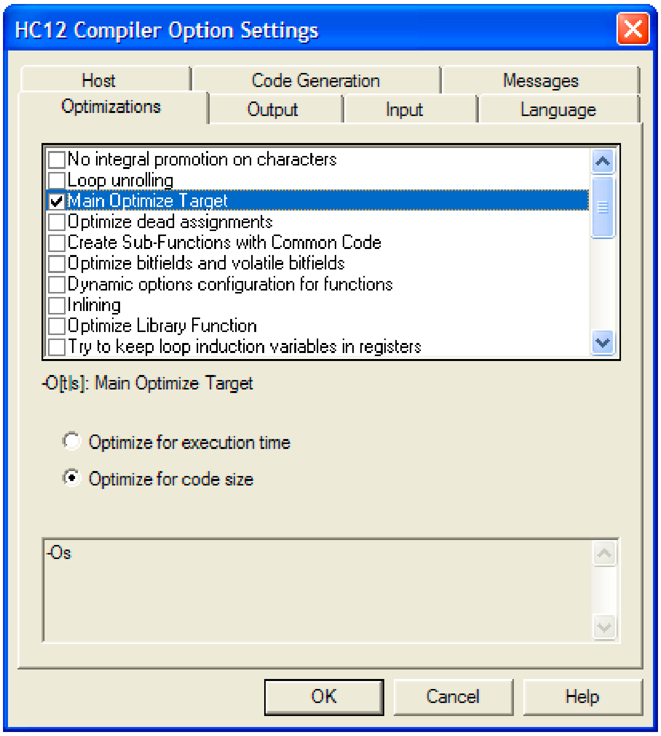
Command line options are not case-sensitive, e.g., `"-Li"` is the same as `"-li"`.

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 HC12 Compiler Option Settings dialog box (Figure 5.1).

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.

Figure 5.1 Option Settings dialog box



- [ ] No integral promotion on characters
- [ ] Loop unrolling
- [✓] Main Optimize Target
- [ ] Optimize dead assignments
- [ ] Create Sub-Functions with Common Code
- [ ] Optimize bitfields and volatile bitfields
- [ ] Dynamic options configuration for functions
- [ ] Inlining
- [ ] Optimize Library Function
- [ ] Try to keep loop induction variables in registers

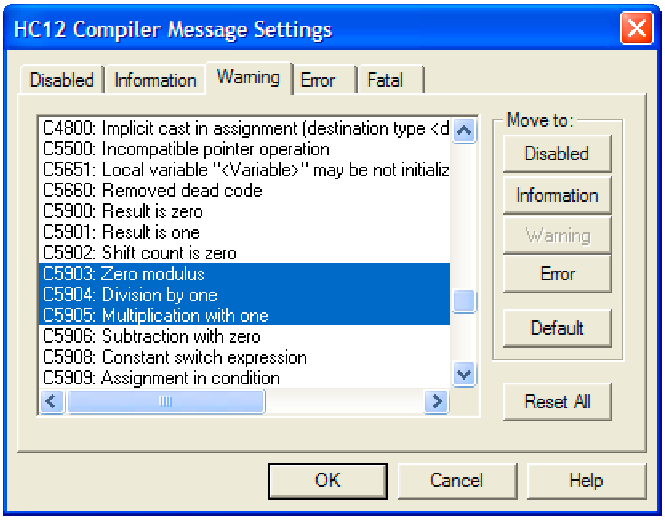
-O[t|s]: Main Optimize Target

Optimize for execution time [✓ Optimize for code size]

```
-Os
```

The HC12 Compiler Message Settings dialog box, shown in Figure 5.2, may also be used to move messages (`-Wmsg` options).

Figure 5.2 HC12 Compiler Message Settings dialog box



C4800: Implicit cast in assignment (destination type <d
C5500: Incompatible pointer operation
C5651: Local variable "<Variable>" may be not initializ
C5660: Removed dead code
C5900: Result is zero
C5901: Result is one
C5902: Shift count is zero
C5903: Zero modulus
C5904: Division by one
C5905: Multiplication with one
C5906: Subtraction with zero
C5908: Constant switch expression
C5909: Assignment in condition

Option Recommendation

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

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

The default configuration enables most 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”
```

Compiler Option Details

Option Groups

Compiler options are grouped by:

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

See Table 5.1.

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.

Table 5.1 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 options related to the output files generation (which kind of file should be generated)
INPUT	Lists options related to the input file

Table 5.1 Compiler option groups (*continued*)

Group	Description
CODE GENERATION	Lists options related to code generation (memory models, float format, ...)
TARGET	Lists options related to the target processor
MESSAGES	Lists options controlling the generation of error messages
VARIOUS	Lists various options
STARTUP	Options which only are specified 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).

Option Scopes

Each option has also a scope. See Table 5.2.

Table 5.2 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.

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 Table 5.3.

Table 5.3 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

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.

Table 5.4 lists the supported modifiers.

Table 5.4 Compiler Option Modifiers

Modifier	Description
%p	Path including file separator
%N	Filename in strict 8.3 format

Table 5.4 Compiler Option Modifiers (*continued*)

Modifier	Description
%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 ‘%’

Examples

For the examples in Listing 5.1, the actual base filename for the modifiers is: `C:\Freescale\my demo\TheWholeThing.myExt`.

Listing 5.1 Examples of compiler option modifiers

(1) `%p` gives the path only with a file separator:
`C:\Freescale\my demo\`

(2) `%N` results in the filename in 8.3 format (that is, the name with only 8 characters):
`TheWhole`

(3) `%n` returns just the filename without extension:
`TheWholeThing`

(4) `%E` gives the extension in 8.3 format (that is, the extension with only 3 characters)
`myE`

(5) `%e` is used for the whole extension:
`myExt`

(6) `%f` gives the path plus the filename:
`C:\Freescale\my demo\TheWholeThing`

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

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

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

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

(9) `%(`*`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
```

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

```
    myfile.txt
```

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

```
    myExt%
```

-!: filenames 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 (8 characters).

Listing 5.2 Example of the cut option, -!

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

-AddIncl: Additional Include File

Group

INPUT

Scope

Compilation Unit

Syntax

`-AddIncl”<fileName>”`

Arguments

`<fileName>`: name of the file that is included

Default

None

Defines

None

Pragmas

None

Description

The specified file is included at the beginning of the compilation unit. It has the same effect as it would if written at the beginning of the compilation unit using double quotes (`“..”`):

```
#include “my headerfile.h”
```

Example

See Listing 5.3 for the `-AddIncl` compiler option that includes the above header file.

Listing 5.3 -AddIncl example

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

See also

-I: Include File Path compiler option

-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.

-Asr: It is assumed that HLI code saves written registers

Group

CODE GENERATION

Scope

Function

Syntax

`-Asr`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option set, the compiler assumes that registers touched in HLI are saved or restored in the HLI code as well. If this option is not set, the compiler will save or restore the H, X, and A registers.

Listing 5.4 Sample source code for the two following examples

```
void test(void) {
  PORT = 4;
  asm {
    LDD   #4
    STD   PORT
  }
  CallMe(4);
}
```

Listing 5.5 Without the -Asr option set (default), we get:

```
0000 c604          [1]     LDAB  #4
0002 87            [1]     CLRA
0003 7c0000        [3]     STD   PORT
0006 cc0004        [2]     LDD   #4
0009 7c0000        [3]     STD   PORT
000c c604          [1]     LDAB  #4
000e 87            [1]     CLRA
000f 060000        [3]     JMP   CallMe
```

With the `-Asr` option set (Listing 5.6), the compiler can assume that the A register is still the same as before the `__asm` block. However, in our example we do NOT save or restore the A register, so the code will be incorrect.

Listing 5.6 With the -Asr option set, we get:

```
0000 c604          [1]     LDAB  #4
0002 87            [1]     CLRA
0003 7c0000        [3]     STD   PORT
0006 cc0004        [2]     LDD   #4
0009 7c0000        [3]     STD   PORT
000c 060000        [3]     JMP   CallMe
```

-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

Listing 5.7 uses the default condition and uses the three least significant bits.

Listing 5.7 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 Listing 5.8.

Listing 5.8 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

-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

0

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 Listing 5.9, it is assumed that you have specified a 3-bit maximum gap, i.e., `-BfaGapLimitBits3`.

Listing 5.9 Bitfield allocation

```
struct {
  unsigned char a: 7;
  unsigned char b: 5;
  unsigned char c: 4;
} B;
```

The compiler allocates struct B with 3 bytes. First, the compiler allocates the 7 bits of a. Then the compiler tries to allocate the 5 bits of b, but this would cross a byte boundary. Because the gap of 1 bit is smaller than the specified gap of 3 bits, b is allocated in the next byte. Then the allocation starts for c. After the allocation of b there are 3 bits left. Because the gap is 3 bits, c is allocated in the next byte. If the maximum gap size were specified to 0, all 16 bits of B would be allocated in two bytes.

Listing 5.10 specifies a maximum size of two bits for a gap.

Listing 5.10 Example where the maximum number of gap bits is 2

```
-BfaGapLimitBits2
```

See also

Bitfield Allocation

-BfaTSR: Bitfield Type-Size Reduction

Group

CODE GENERATION

Scope

Function

Syntax

```
-BfaTSR(ON|OFF)
```

Arguments

`ON`: Enable Type-Size Reduction
`OFF`: Disable Type-Size Reduction

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

Listing 5.11 and Listing 5.12 demonstrate the effects of `-BfaTSRoff` and `-BfaTSRon`, respectively.

Listing 5.11 -BfaTSRoff

```
struct{
  long b1:4;
  long b2:4;
} myBitfield;

31                      7  3  0
--------------------------------
|########################|b2|b1|  -BfaTSRoff
--------------------------------
```

Listing 5.12 -BfaTSRon

```
7   3    0
----------
|b2 | b1 |  -BfaTSRon
----------
```

Example

```
-BfaTSRon
```

See also

Bitfield Type Reduction

-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 3 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 stuff 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”

-Cc: Allocate Constant 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

In the HIWARE Object-file Format, variables declared as `const` are treated just like any other variable, unless the `-Cc` command-line option was used. In that circumstance, the `const` objects are put into the `ROM_VAR` segment, which is then assigned to a ROM section in the Linker parameter file (please see the *Linker* section in the Build Tools manual).

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 (please 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 (refer to the *Linker* section of the Build Tools manual for more information).

NOTE This option is useful only for HIWARE object-file formats. 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

Listing 5.13 shows how the `-Cc` compiler option affects the `SECTIONS` segment of a PRM file (HIWARE object-file format only).

Listing 5.13 -Cc example (HIWARE format only)

```
SECTIONS
  MY_ROM  READ_ONLY     0x1000 TO 0x2000
PLACEMENT
  DEFAULT_ROM, ROM_VAR  INTO MY_ROM
```

See also

Segmentation
Linker section in the Build Tools manual
-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format option
#pragma INTO_ROM: Put Next Variable Definition into ROM

-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 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 Listing 5.14 for a prm file about how to place the sections mentioned in the Table 5.5.

Table 5.5 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 CodeWarrior please refer to the technical note TN 234. Table 5.6 indicates how space modifiers are mapped for the HC(S)12:

Table 5.6 Cosmic Space modifier mapping for the HC12

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

See Listing 5.14 for an example of the -Ccx compiler option.

Listing 5.14 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, TN 234

-Cf: Float IEEE32, doubles IEEE64

Group

CODE GENERATION

Scope

Application

Syntax

Cf

Arguments

None

Default

By default, float and doubles are IEEE32

Defines

```
__FLOAT_IS_IEEE32__
__DOUBLE_IS_IEEE64__
__LONG_DOUBLE_IS_IEEE64__
__LONG_LONG_DOUBLE_IS_IEEE64__
```

Pragmas

None

Description

This option sets the standard type `float` to the IEEE32 format and all double types (`double`, `long double`, `long long double`) to the IEEE64 format.

This option is the same as `-Tf4d8Ld8LLd8`.

Example

```
-Cf
```

See also

-T: Flexible Type Management compiler option

-Ci: Tri- and Bigraph 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 Table 5.7.

Table 5.7 Keyword Alternatives for Unavailable Tokens

Bigraph		Trigraph		Additional Keyword	
<%	}	??=	#	and	&&
%>	}	??/	\	and_eq	&=
<:	[	??’	^	bitand	&
:>	]	??(	[	bitor	|
%:	#	??)	]	compl	~

Table 5.7 Keyword Alternatives for Unavailable Tokens (*continued*)

Bigraph		Trigraph		Additional Keyword	
%:%:	##	??!	|	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 Listing 5.15 shows the use of trigraphs, bigraphs, and the additional keywords with the corresponding ‘normal’ C-source.

Listing 5.15 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
#defin 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;
    }
    }
}
```

-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
```

-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 (Listing 5.16) be extended to `int`. The `-Cni` option disables this extension and saves additional RAM.

Example

See Listing 5.16 for an example of “no integer promotion.”

Listing 5.16 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 ... are treated as integer. If you want them treated as characters, they must be casted (Listing 5.17).

Listing 5.17 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.

-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 (Listing 5.18).

Listing 5.18 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 compiler option

-CpDIRECT: DIRECT Register Value

Group

CODE GENERATION

Scope

Application

Syntax

```
-CpDIRECT<hexAddr>
```

Arguments

`<hexAddr>`: Start address of direct window

Default

The Compiler assumes that the DIRECT register contains 0.

Defines

```
__DIRECT_ADR__=<adr>
```

Pragmas

None

Description

This option only has an effect for the HCS12X (if option `-CpuHCS12X` is specified).

For the HC12/HCS12 families, all direct accesses were using accessing the address range from `0x0000` to `0x00FF`. In this range, the most often resource could be mapped to benefit from the shorter direct addressing mode compared to the extended addressing mode.

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

Because of this change, the compiler needs to know which part of the address space is accessible through with the direct addressing mode.

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

Note that this knowledge is only necessary to optimize this if only the address is known. Variables allocated in a `__SHORT_SEG` section are not affected by this option.

Example

```
-CpDIRECT8192
*((int*)0x2002)=3;
```

Generates:

```
0000 c603         LDAB  #3
0002 87           CLRA
0003 5c02         STD   2
```

See also

Compiler options:

- -CpDPAGE: Specify DPAGE Register
- -CpEPAGE: Specify EPAGE Register
- -CpGPAGE: Specify GPAGE Register
- -CpPPAGE: Specify PPAGE Register
- -CpRPAGE: Specify RPAGE Register

-CpDPAGE: Specify DPAGE Register

Group

CODE GENERATION

Scope

Application

Syntax

`-CpDPAGE[=(<hexAddr>|RUNTIME)]`

Arguments

`<hexAddr>`: address of the DPAGE register in hex format (e.g., `0x34`)

`RUNTIME`: if runtime routine must be used

Default

By default, the Compiler assumes `0x34` for `<hexAddr>`

Defines

`__DPAGE__`

`__NO_DPAGE__`

`__DPAGE_ADR__ = hexAddr`

Pragmas

None

Description

Only the HC12 A4 derivative has a DPAGE register. See the Backend chapter for details.

NOTE The RUNTIME argument for this option is not available when in HCS12X or HCS12XE mode.

Example

```
-CpDPAGE=RUNTIME
```

See also

Compiler options:

- -CpEPAGE: Specify EPAGE Register
- -CpGPAGE: Specify GPAGE Register
- -CpPPAGE: Specify PPAGE Register
- -CpRPAGE: Specify RPAGE Register

-CpEPAGE: Specify EPAGE Register

Group

CODE GENERATION

Scope

Application

Syntax

`-CpEPAGE[=(<hexAddr>|RUNTIME)]`

Arguments

`<hexAddr>`: address of the EPAGE register in hex format (e.g., `0x17`)

`RUNTIME`: if runtime routine must be used

Default

Depending on the `-Cpu` option, `0x36` is used for an HC12 A4 or `0x17` for an HCS12X.

Defines

`__EPAGE__`

`__NO_EPAGE__`

`__EPAGE_ADR__ = hexAddr`

Pragmas

None

Description

The HC12 A4 derivative and the HCS12X family have an EPAGE register. See Backend for details.

NOTE The RUNTIME argument for this option is not available when in HCS12X or HCS12XE mode.

Example

```
-CpEPAGE=0x17
```

See also

Compiler options:

- -CpDPAGE: Specify DPAGE Register
- -CpGPAGE: Specify GPAGE Register
- -CpPPAGE: Specify PPAGE Register
- -CpRPAGE: Specify RPAGE Register

-CpGPAGE: Specify GPAGE Register

Group

CODE GENERATION

Scope

Application

Syntax

`-CpGPAGE[=(<hexAddr>)].`

Arguments

`<hexAddr>`: address of the `GPAGE` register in hex format (e.g., `0x10`)

Default

By default, the Compiler assumes `0x10` for `<hexAddr>`

Defines

`__GPAGE__`

`__NO_GPAGE__`

`__GPAGE_ADR__ = hexAddr`

Pragmas

None

Description

Only HCS12X family members have a GPAGE register and support GPAGE access.

GPAGE accesses are performed with the special G load or store instructions and is therefore different from the other page accesses which all are using some address window in the logical address space.

GPAGE accesses are using global addresses and are performed in the global address space.

See the Backend chapter for details.

Example

```
-CpGPAGE=0x36
```

See also

Compiler options:

- -CpDPAGE: Specify DPAGE Register
- -CpEPAGE: Specify EPAGE Register
- -CpPPAGE: Specify PPAGE Register
- -CpRPAGE: Specify RPAGE Register

-CpPPAGE: Specify PPAGE Register

Group

CODE GENERATION

Scope

Application

Syntax

`-CpPPAGE[=(<hexAddr>|RUNTIME)].`

Arguments

`<hexAddr>`: address of the PPAGE register in hex format (e.g., `0x30`)

`RUNTIME`: if runtime routine must be used

Default

Depending on the `-Cpu` option, `0x35` is used for an HC12 A4 or `0x30` for an HCS12 or HCS12X.

Defines

`__PPAGE__`

`__NO_PPAGE__`

`__PPAGE_ADR__ = hexAddr`

Pragmas

None

Description

The `PPAGE` value specified with this option is only used for data paging. For code banking with a `CALL` instruction, this option is not required. See Backend for details.

NOTE The RUNTIME argument for this option is not available when in HCS12X or HCS12XE mode.

Example

```
-CpPPAGE=0x30
```

See also

Compiler options:

- -CpDPAGE: Specify DPAGE Register
- -CpEPAGE: Specify EPAGE Register
- -CpGPAGE: Specify GPAGE Register
- -CpRPAGE: Specify RPAGE Register

-CpRPAGE: Specify RPAGE Register

Group

CODE GENERATION

Scope

Application

Syntax

`-CpRPAGE[=(<hexAddr>|RUNTIME)].`

Arguments

`<hexAddr>`: address of the RPAGE register in hex format (e.g., `0x16`)

`RUNTIME`: if runtime routine must be used

Default

`0x16 for <hexAddr>`

Defines

`__RPAGE__`

`__NO_RPAGE__`

`__RPAGE_ADR__ = hexAddr`

Pragmas

None

Description

See the Backend chapter for details.

NOTE The RUNTIME argument for this option is not available when in HCS12X or HCS12XE mode.

Example

```
-CpRPAGE=0x16
```

See also

Compiler options:

- -CpDPAGE: Specify DPAGE Register
- -CpEPAGE: Specify EPAGE Register
- -CpGPAGE: Specify GPAGE Register
- -CpPPAGE: Specify PPAGE Register

-Cpu: Generate code for specific HC(S)12 families

Group

CODE GENERATION

Scope

Application

Syntax

`-Cpu(CPU12|HCS12|HCS12X|HCS12XE)`

Arguments

`CPU12`: Generate code for a CPU12.

`HCS12`: Generate code for an HCS12.

`HCS12X`: Generate code for an HCS12X.

`HCS12XE`: Generate code for an HCS12XE

Default

The Compiler generates code for a CPU12.

Defines

`__HC12__`: always defined

`__HCS12__`: defined for the `-CpuHCS12` and `-CpuHCS12X` options

`__HCS12X__`: defined for the `-CpuHCS12X` option

`__HCS12XE__`: defined for the `-CpuHCS12XE` option

Pragmas

None

Description

This option controls for which family the code should be generated. The two choices `-CpuHCS12` and `-CpuCPU12` generate almost identical code which is completely compatible. The HCS12 and the CPU12 cores only differ in their execution timings and for PC relative `MOVB` or `MOVW` operands, which are not used by C code.

The `-CpuHCS12X` option allows the use of the new instructions of the HCS12X as well. The code generated is incompatible to an HCS12 or CPU12 core.

Code generated for the HCS12 or CPU12 can be executed on an HCS12X, but does not utilize the advantages of the new architecture. Mixing modules compiled for the HCS12X and the HC12 or HCS12 is possible but not recommended. Especially the representation of `__far` data pointers is different.

Switching to or from the `-CpuHCS12X` code generation requires the following adaptations in a project:

- Use the `-CpuHCS12X` option for both the compiler and the assembler.
- Use the correct ANSI library.

The libraries for the HCS12X contain an `X` after ANSI in their filenames.

The HCS12XE is an extension of the HCS12X. The HCS12XE supports mapping to RAM area 0x4000–0x7FFF, which results in different mapping to logical and global addresses.

See HC(S)12 Backend for details.

Example

```
-CpuHCS12X
```

-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 Listing 5.19 declares two structs, each of which has a `const` member.

Listing 5.19 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 would cause an error message.

-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 8 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 Listing 5.20 is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’).

Listing 5.20 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% will produce 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 Search Switch Tables
- -CswMinLF: Minimum Load Factor for Switch Tables

-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 Search Switch Tables
- -CswMaxLF: Maximum Load Factor for Switch Tables

-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 Listing 5.21 is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’).

Listing 5.21 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 Search Switch Tables
- -CswMaxLF: Maximum Load Factor for Switch Tables

-CswMinSLB: Minimum Number of Labels for Search Switch 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

-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 5.22 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 Listing 5.23.:

Listing 5.23 Transformation of the for Loop in Listing 5.22

```
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 5.24 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:

`i=1;`

because the loop body is never executed.

Listing 5.25 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.

-CVolWordAcc: Do not reduce volatile word accesses

Group

CODE GENERATION

Scope

Function

Syntax

```
-CVolWordAcc
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Forces the compiler to generate a word access on 16 bit large volatile integral types. Typical application is the access to 16 bit large I/O registers. The option has negative effect on code efficiency because bit set, bit clear and bit test operations are not used (see example below).

Example

```
volatile int i;
void foo(void) {
  if ((i & 0x200) != 0) {
    ff();
  }
}
```

with option `-CVolWordAcc`

```
  LDD   i
  CLRB
  ANDA  #2
  TBEQ  D,exit
  JSR   ff
exit:
  RTS
```

without option `-CVolWordAcc`

```
  BRCLR i,#2,exit
  JSR   ff
exit:
  RTS
```

See also

None

-Cx: No 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.

-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.

Listing 5.26 DEBUG macro definition.

```
-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: 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.

-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 Listing 5.27 and Listing 5.28 for examples using -Ec conversions.

Listing 5.27 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 5.28 Assigning a value to a “constant” pointer

```
-Ec

void foo(const int *p){
  *p = 0; // Some Compilers do not issue an error.
```

-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 (see Using Special Modifiers).

Default

The default filename is `%f.e%e`. A file named ‘`foo.c`’ creates an encrypted file named ‘`foo.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

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

encrypts the ‘`foo.c`’ file using the 1234567 key to the ‘`foo.ec`’ file and the ‘`foo.h`’ file to the ‘`foo.eh`’ file.

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

```
foo.ec
```

See also

-Ekey: Encryption Key

-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

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

encrypts the ‘`foo.c`’ file using the 1234567 key.

See also

-Eencrypt: Encrypt Files

-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

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

Group

OUTPUT

Scope

Application

Syntax

```
-F(h|1|1o|2|2o|6|7)
```

Arguments

`h`: HIWARE object-file format
`1`: ELF/DWARF 1.1 object-file format
`1o`: compatible ELF/DWARF 1.1 object-file format
`2`: ELF/DWARF 2.0 object-file format
`2o`: compatible ELF/DWARF 2.0 object-file format
`6`: strict HIWARE V2.6 object-file format
`7`: strict HIWARE V2.7 object-file format

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

Default

```
-F2
```

Defines

```
__HIWARE_OBJECT_FILE_FORMAT__
__ELF_OBJECT_FILE_FORMAT__
```

Pragmas

None

Description

The Compiler writes the code and debugging info after compilation into an object file.

The Compiler uses a HIWARE-proprietary object-file format when the `-Fh`, `-F6`, or `-F7` options are set.

The HIWARE Object-file Format (`-Fh`) has the following limitations:

- The type char is limited to a size of 1 byte.
- Symbolic debugging for enumerations is limited to 16-bit signed enumerations.
- No zero bytes in strings are allowed (a zero byte marks the end of the string).

The HIWARE V2.7 Object-file Format (`-F7` option) has some limitations:

- The type char is limited to a size of 1 byte.
- Enumerations are limited to a size of 2 bytes and have 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 (a zero byte marks the end of the string).

The Compiler produces an ELF/DWARF object file when the `-F1` or `-F2` options are set. This object-file format may also be supported by other Compiler vendors.

In the Compiler ELF/DWARF 2.0 output, some constructs written in previous versions 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.

The `-F1o` option is only supported if the target supports the ELF/DWARF 1.1 format. This option is only used with older debugger versions as a compatibility option. This option may be discontinued in the future. It is recommended you use `-F1` instead.

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.

-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.

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

Example

Listing 5.29 lists the short list options.

Listing 5.29 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
```

-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

-Ica: Implicit Comments in HLI-ASM Instructions

Group

LANGUAGE

Scope

Function

Syntax

```
-Ica
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Comments in HLI (High-Level Inline) Assembler are either normal High-Level Language comments (e.g., using ANSI-C comments `/* */` or C++ comments `//`), or HLI comments beginning with ‘`;`’.

If this option is enabled, the Compiler handles all text as comments after a complete assembly statement. It is not necessary to start an HLI comment with a special token (‘`;`’, ‘`/*`’ or ‘`//`’). This is useful when compiling assembly source from other assemblers that allow this option.

Example

```
-Ica
```

-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

-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

-Lasmc: Configure Listing File

-Lasmc: Configure Listing File

Group

OUTPUT

Scope

Function

Syntax

```
-Lasmc[={a|c|i|s|h|p|e|v|y}]
```

Arguments

`a`: Do not write the address in front of every instruction
`c`: Do not write the hex bytes of the instructions
`i`: Do not write the decoded instructions
`s`: Do not write the source code
`h`: Do not write the function header
`p`: Do not write the source prolog
`e`: Do not write the source epilog
`v`: Do not write the compiler version
`y`: Do not write cycle information

Default

All printed together with the source

Defines

None

Pragmas

None

Description

The -Lasmc option configures the output format of the listing file generated with the -Lasm: Generate Listing File option. The addresses, the hex bytes, and the instructions are selectively switched off.

The format of the listing file has layout shown in Listing 5.30. The letters in brackets ([]) indicate which suboption may be used to switch it off:

Listing 5.30 -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
```

-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, 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

Listing 5.31 is an example which lists the contents of a file containing define directives.

Listing 5.31 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

-Li: List of Included Files

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

Listing 5.32 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 5.32 Display contents of a file when include directives are present

```
-Li
```
If the source file is: `C:\myFiles\b.c:`
```
/* C:\myFiles\b.c */
```

```
#include <string.h>
```

Then the generated file is:

```
C:\myFiles\b.c :\
C:\Freescale\lib\targetc\include\string.h \
C:\Freescale\lib\targetc\include\libdefs.h \
C:\Freescale\lib\targetc\include\hidef.h \
C:\Freescale\lib\targetc\include\stddef.h \
C:\Freescale\lib\targetc\include\stdtypes.h
```

See also

-Lm: List of Included Files in Make Format compiler option

-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

-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

-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

-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

-Ll: Statistics about Each Function

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), stack 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

Listing 5.33 is an example where the use of the `-L1` compiler options allows statistical information to be added to the end of an output listing file.

Listing 5.33 Statistical information appended to an assembler listing

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

void Func2(void) {
}
```

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

```
foo.c Func1 -Ll=mylog.txt   11  4 0.055000
foo.c Func2 -Ll=mylog.txt       1  0 0.001000
```

-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

Listing 5.34 is an example where the `-Lm` option generates a make file containing include directives.

Listing 5.34 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
-Lo: Object File List

-LmCfg: Configuration of List of Included Files in Make Format

Group

OUTPUT

Scope

Compilation Unit

Syntax

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

Arguments

`i`: Write path of included files
`l`: Use line continuation
`m`: Write path of main file
`o`: Write path of object file
`u`: Update information

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 Listing 5.35:

Listing 5.35 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 Listing 5.36:

Listing 5.36 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 (Listing 5.37):

Listing 5.37 i suboption

```
b.o: b.c C:\Freescale\lib\include\stdio.h
C:\Freescale\lib\include\stddef.h C:\Freescale\lib\include\stdarg.h
C:\Freescale\lib\include\ C:\Freescale\lib\include\string.h
```

The `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
- -Lo: Object File List
- -Lm: List of Included Files in Make Format

-Lo: Object File List

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
- -Lm: List of Included Files in Make Format

-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

-LpCfg: Preprocessor Output configuration

Group

OUTPUT

Scope

Compilation Unit

Syntax

`-LpCfg[={c|f|l|s}]`

Arguments

`c`: Do not generate line comments
`e`: Generate empty lines
`f`: Filenames with path
`l`: Generate #line directives in preprocessor output
`m`: Do not generate filenames
`s`: Maintain spaces

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 Table 5.8.

Table 5.8 Effects of Source and Line Information Format Control Arguments

Argument	on	off
"c"	#line 1 #line 10	/* 1 */ /* 2 */ /* 10 */

"e"	int j;
	
	int i;
		int j;
		
		int i;

"f"	C:\Freescale\include\stdlib.h	stdlib.h
"l"	#line 1 "stdlib.h"	/**** FILE 'stdlib.h' */
"m"		/**** FILE 'stdlib.h' */

"s"	/* 1 */ int f(void) {
	/* 2 */ return 1;
	/* 3 */ }
		/* 1 */ int f (void) {
		/* 2 */ return 1 ;
		/* 3 */ }

all	#line 1 "C:\Freescale\include\stdlib.h" #line 10	/**** FILE 'stdlib.h' */ /* 1 */ /* 2 */ /* 10 */

Example

```
-Lpcfg
-Lpcfg=lfs
```

See also

-Lp: Preprocessor Output

-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

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

Group

CODE GENERATION

Scope

Application

Syntax

`-M(s|b|l)`

Arguments

`s`: small memory model
`b`: banked memory model
`l`: large memory model

Default

`-Ms`

Defines

```
__SMALL__
__BANKED__
__LARGE__
```

Pragmas

None

Description

See the Backend chapter for details (Memory Models).

Example

`-Ms`

-Map: Define mapping for memory space 0x4000-0x7FFF

Group

CODE GENERATION

Scope

Application

Syntax

```
-Map(RAM|FLASH|Exernal)
```

Arguments

`RAM`: maps accesses to 0x4000–0x7FFF to 0x0F_C000–0x0F_FFFF in the global memory space (RAM area).

`FLASH`: maps accesses to 0x4000–0x7FFF to 0x7F_4000–0x7F_7FFF in the global memory space (FLASH).

`External`: maps accesses to 0x4000–0x7FFF to 0x14_4000–0x14_7FFF in the global memory space (external access).

Default

`FLASH`

Defines

None

Pragmas

None

Description

This option sets the memory mapping for addresses between 0x4000 and 0x7FFF for HCS12XE. This mapping is determined by the MMC control register (the `ROMHM` and `RAMHM` bits) and the compiler must be aware of the current setting to correctly perform address translations.

Example

```
-MapRAM
```

-N: Display Notify Box

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 (please see *Make Utility*) because the Compiler waits for you to acknowledge the message, thus suspending make file processing. The `N` stands for “Notify”.

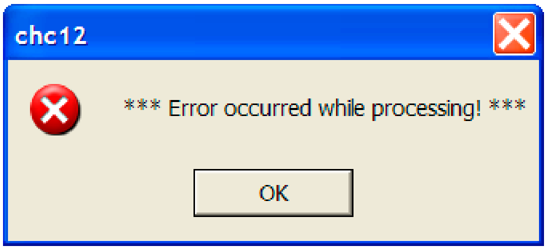
This feature is useful for halting and aborting a build using the Make Utility.

Example

```
-N
```

If an error occurs during compilation, a dialog box similar to the one in Figure 5.3 appears.

Figure 5.3 Alert Dialog Box



chc12

*** Error occurred while processing! ***

-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
```

-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 (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format
- -NoPath: Strip Path Info

-NoEnv: Do not Use Environment

Group

STARTUP. This option cannot be specified interactively.

Scope

None

Syntax

```
-NoEnv
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option can only be specified at the command line while starting the application. It can not be specified in any other way, including via the default.env file, the command line, or processes.

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

Example

```
compiler.exe -NoEnv
```

Use the compiler executable name instead of “compiler”.

See also

Local Configuration File (usually project.ini)

-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

-O (-Os, -Ot): Main Optimization Target

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-O(s|t)`

Arguments

`s`: Optimization for code size (default)
`t`: Optimization for execution speed

Default

`-Os`

Defines

`__OPTIMIZE_FOR_SIZE__`
`__OPTIMIZE_FOR_TIME__`

Pragmas

None

Description

There are various 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 will “trade” slower-smaller code for 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
```

-Obfv: Optimize Bitfields and Volatile Bitfields

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Obfv`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Optimize bitfields as well as bitfields declared as volatile. The compiler is allowed to change the access order or to combine many accesses to one, even if the bitfields are declared as volatile.

Example

Listing 5.38 contains bitfields to be optimized with the `-Obfv` compiler option.

Listing 5.38 Bitfields example

```
volatile struct {
  unsigned int b0:1;
  unsigned int b1:1;
  unsigned int b2:1;
} bf;
void foo(void) {
  bf.b0 = 1;  bf.b1 = 1;  bf.b2 = 1;
}
```

Listing 5.39 shows the effect of the `-Obfv` option.

Listing 5.39 Results of using the -Obfv option

```
using -Obfv:
BSET  bf,#7

without -Obfv:
BSET  bf,#1
BSET  bf,#2
BSET  bf,#4
```

-ObjN: Object filename 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 environment variable

-Oc: Common Subexpression Elimination (CSE)

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-Oc
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Performs common subexpression elimination (CSE). The code for common subexpressions and assignments is generated only once. The result is reused. Depending on available registers, a common subexpression may produce more code due to many spills.

NOTE When the CSE is switched on, changes of variables by aliases may generate incorrect optimizations.

This option is disabled and present only for compatibility reasons for the Freescale HC(S)12

Example

```
-Oc
```

Listing 5.40 is an example where the use of the CSE compiler option causes incorrect optimizations. But, no matter, because this option is not enabled any longer in any event for the HC(S)12.

Listing 5.40 Example where CSE may produce incorrect results

```
void main(void) {
  int x;
  int *p;
  x = 7;  /* here the value of x is set to 7 */
  p = &x;
  *p = 6;  /* here x is set to 6 by the alias *p */
  /* here x is assumed to be equal to 7 and
     Error is called */
  if(x != 6) Error();
}
```

NOTE This error does not occur if `x` is declared as `volatile`.

-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 Listing 5.41:

Listing 5.41 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 No options with scope Application or Compilation Unit (as memory model, float or double format, or object-file format) or options for the whole compilation unit (like inlining or macro definition) should be specified in this option. 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"
```

-Of or -Onf: Create Sub-Functions with Common Code

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Onf`

Arguments

None

Default

`-Of` default or with -Os; -Onf with -Ot”

Defines

None

Pragmas

None

Description

This option performs the reverse job of inlining. It detects common code parts in the generated code. The common code is moved to a different place, and all occurrences are replaced with a `JSR` to the moved code. At the end of the common code, an `RTS` instruction is inserted. All SP usages are increased by an address size. This optimization takes care of stack allocation, control flow, and of functions having arguments on the stack. Also, inline assembler code is never treated as common code.

Example

Consider the following function in Listing 5.42:

Listing 5.42 Function example

```
void f(int);
void g(void);
void h(void);
void main(void) {
  f(1);  f(2);  f(3);
  h();
  f(1);  f(2);  f(3);
  g();
  f(1);  f(2);
}
```

The compiler first detects that "`f(1); f(2); f(3);`" occurs twice and places this code separately.

The two code patterns are replaced by a call to the moved code.

This situation can be thought of as the following non-C pseudo code (C does not support local functions):

```
void main(void) {
  void tmp0(void) {
  f(1);  f(2);  f(3);
  }
  tmp0();
  h();
  tmp0();
  g();
  f(1);  f(2);
}
```

In a next step, the Compiler detects that the code "f(1); f(2);" also occurs twice. The Compiler generates a second internal function:

```
void main(void) {
  void tmp1(void) {
    f(1);  f(2);
  }
  void tmp0(void) {
    tmp1();  f(3);
  }
  tmp0();
  h();
  tmp0();
  g();
```

```
   tmp1();
}
```

The new code of the `tmp1` function (actually `tmp1` is not really a function, but it is a part of `main()`) is called once directly from main, and once indirectly by using `tmp0`. These two call chains use a different amount of the stack. Because of this situation, it is not always possible to generate correct debug information. For the local function tmp1, the compiler cannot state both possible SP values. It will only state one of them. While debugging the other state, local variables and the call chain are declared invalid in the debugger. The compiler notes this situation and issues the message:

"C12056: SP debug info incorrect because of optimization or inline assembler"

Tips

Switch off this optimization to debug your application. The common code makes the control flow more complicated. Also, the debugger cannot distinguish two distinct usages of the same code. Setting a breakpoint in common code stops the application and every use of it. It will also stop the local variables and the call frame if they are not displayed correctly, as explained above.

Switching off this optimization achieves faster code. For code density, there are only a few cases where the code gets worse. This situation may only occur when other optimizations (such as branch tail merging or peepholing) cannot find a pattern after this optimization occurs.

See also

Message "C12056: SP debug info incorrect because of optimization or inline assembler"

-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

-Oilib: Optimize Library Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

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

Arguments

`<arguments>` are one or multiple of following suboptions:
`a`: inline calls to the `strcpy()` function
`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()

`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 for 32-bit shifts, the arrays _PowOfTwo_16 and _PowOfTwo_32 are used. These constant arrays contain the values 1, 2, 4, 8... . They are declared in `hidef.h`. 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);
}
```

is translated similar to the following function:

```
void g(void) {
  s2 = s;
  while(*s2++ = *ct++);
}
```

See also

-Oi: Inlining
Message C5920

-Ol: Try to Keep Loop Induction Variables in Registers

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Ol<number>`

Arguments

`<number>`: number of registers to be used for induction variables

Default

None

Defines

None

Pragmas

None

Description

Try to maintain `<number>` loop induction variables in registers. Loop induction variables are variables read and written within the loop (e.g., loop counter). The Compiler tries to keep loop induction variables in registers to reduce execution time, and sometimes also code size. This option sets the number of loop induction variables the Compiler is allowed to keep in registers. The range is from 0 (no variable) to infinity. If this option is not given, the Compiler takes the optimal number (code density). Like the option -or, this option may increase code size (spill and merge code) if too many loop induction variables are specified.

NOTE Disable this option (with `-Ol0`) if there are problems when debugging your code. This optimization could increase the complexity of code debugging on a High-Level Language level.

The example in Listing 5.43 is used in Listing 5.44 and in Listing 5.45.

Listing 5.43 Example (abstract code)

```
void main(char *s) {
  do {
    *s = 0;
  } while (*++s);
}
```

Listing 5.44 shows pseudo disassembly with the `-Ol0` option:

Listing 5.44 With the -Ol0 option (no optimization, pseudo code)

```
loop:
  LD  s, Reg0
  ST  #0, [Reg0]
  INC Reg0
  ST  Reg0, s
  CMP [Reg0],#0
  BNE loop
```

Listing 5.45 shows pseudo disassembly without the `-Ol` option (i.e., optimized) where the load and stores from or to variable s disappear.

Listing 5.45 Without option (optimized, pseudo assembler)

```
loop:
  ST  #0, s
  INC s
  CMP s,#0
  BNE loop
```

Example

`-Ol1`

-Ona: Disable Alias Checking

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Ona`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Variables that may be written by a pointer indirection or an array access are redefined after the optimization. This option prevents the Compiler from doing this redefinition, which may allow you to reuse already-loaded variables or equivalent constants. Use this option only if you are sure you will have no real writes of aliases to a memory location of a variable.

Example: do not compile with `-Ona`.

```
void main(void) {
  int a = 0, *p = &a;

  *p = 1; // real write by the alias *p
  if (a == 0) Error(); // if -Ona is specified,
  // Error() is called!
}
```

Example

```
-Ona
```

-OnB: Disable Branch Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

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

Arguments

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

`a`: Short BRA optimization
`b`: Branch JSR to BSR optimization
`l`: Long branch optimization
`t`: Branch tail optimization

Default

None

Defines

None

Pragmas

None

Description

See Backend for details.

Example

```
-OnB
```

Disables all branch optimizations

-Onbf: Disable Optimize Bitfields

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Onbf`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

A sequence of bitfield assignments with constants is not combined if you use `-Onbf`. This option simplifies debugging and makes the code more readable.

Example

Listing 5.46 Example bitfield definition

```
struct {
  b0:1;
  b1:1;
  b2:1;
} bf;

void main(void) {
  bf.b0 = 0;
  bf.b1 = 0;
  bf.b2 = 0;
}

without -Onbf: (pseudo intermediate code)
  BITCLR  bf, #7  // all 3 bits (the mask is 7)
                 // are cleared

with -Onbf: (pseudo intermediate code)
  BITCLR  bf, #1  // clear bit 1 (mask 1)
  BITCLR  bf, #2  // clear bit 2 (mask 2)
  BITCLR  bf, #4  // clear bit 3 (mask 4)
```

Example

```
-Onbf
```

-Onbt: Disable ICG Level Branch Tail Merging

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Onbt`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The ICG level branch tail merging is switched off leading to more readable code and simplified debugging.

The example in Listing 5.47 is used in Listing 5.48 and in Listing 5.49.

Listing 5.47 Example function

```
void main(void) {

  if(x > 0) {
    y = 4;
  } else {
  y = 9;
  }
}
```

Without `-Onbt`, the above example disassembles as in Listing 5.48

Listing 5.48 Case (1) without -Onbt: (pseudo intermediate code)

```
             CMP   x, 0
             BLE   else_label

             LOAD  reg, #4
             BRA   branch_tail

else_label:  LOAD  reg, #9
branch_tail: STORE y, reg
go_on:       ...
```

With the `-Obnt` compiler option, Listing 5.47 disassembles as in Listing 5.49.

Listing 5.49 Case (2) with -Onbt: (pseudo intermediate code)

```
             CMP   x, 0
             BLE   else_label

             LOAD  reg, #4
             STORE y, reg
             BRA   go_on

else_label:  LOAD  reg, #9
             STORE y, reg
go_on:       ...
```

Example

```
-Onbt
```

-Onca: Disable any Constant Folding

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Onca`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Disables any constant folding over statement boundaries. This option prevents the Compiler from folding constants. All arithmetical operations are coded. This option must be set when the library functions, `setjmp()` and `longjmp()`, are present. If this option is not set, the Compiler makes wrong assumptions as in the example in Listing 5.50:

Listing 5.50 Example with "if condition always true"

```
void main(void) {
  jmp_buf env;
  int k = 0;
  if (setjmp(env) == 0) {
    k = 1;
    longjmp(env, 0);
    Err(1);
  } else if (k != 1) {/* assumed always TRUE */
    Err(0);
  }
}
```

Example

```
-Onca
```

-Oncn: Disable Constant Folding in case of a New Constant

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Oncn`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Disables any constant folding in the case of a new constant. This option prevents the Compiler from folding constants if the resulting constant is new.
The option only has an effect for processors where a constant is difficult to load (e.g., RISC processors).

Listing 5.51 Example (pseudo code)

```
void main(void) {
  int a = 1, b = 2, c, d;

  c = a + b;
  d = a * b;
}

Case (1) without the -Oncn  option(pseudo code):
```

```
a MOVE 1
b MOVE 2
c MOVE 3

Case (2) with the -Oncn option (pseudo code):
a MOVE 1
b MOVE 2
c ADD a,b
d MOVE 2
```

The constant 3 is a new constant that does not appear in the source. The constant 2 is already present, so it is still propagated.

Example

```
-Oncn
```

-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 HIWARE format, the object-file format permits the Compiler to remove single assignments in a structure or array initialization. In the ELF format, it is optimized 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, the initialization with 0 can only be optimized in the HIWARE format. In ELF it is not possible to separate them from the initialization with 1.

-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 to have unoptimized code.

-One: Disable any low-level Common Subexpression Elimination

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
-One
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prevents the Compiler from reusing common subexpressions, such as array indexes and array base addresses. The code size may increase. The low-level CSE does not have the alias problems of the frontend CSE and is therefore switched on by default.

The two CSE optimizations do not cover the same cases. The low-level CSE has a finer granularity but does not handle all cases of the frontend CSE.

Use this option only to generate more readable code for debugging.

Listing 5.52 Example: (abstract code)

```
void main(int i) {
  int a[10];
  a[i] = a[i-1];
}
```

Listing 5.53 shows the disassembled code without using the `-One` option, whereas Listing 5.54 shows the result of not using the `-One` option.

Listing 5.53 Case (1) without the -One option (optimized)

```
tmp1     LD     i
tmp2     LSL    tmp1,#1
tmp3     SUB    tmp2,#2
tmp4     ADR    a
tmp5     ADD    tmp3, tmp4
tmp6     LD     (tmp5)
2(tmp5)  ST     tmp6
```

Listing 5.54 Case (2) using -One (not optimized, readable)

```
tmp1     LD     i
tmp2     LSL    tmp1,#1
tmp3     SUB    tmp2,#2
tmp4     ADR    a
tmp5     ADD    tmp3,tmp4
tmp6     LSL    tmp1,#1    ;calculated twice
tmp7     ADR    a          ;calculated twice
tmp8     ADD    tmp6,tmp7
tmp9     LD     (tmp5)
(tmp8)   ST     tmp9
```

Example

```
-One
```

-OnP: Disable Peephole Optimization

Group

OPTIMIZATIONS

Scope

Function

Syntax

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

Arguments

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

`a`: Disable `LEAS` to `PUSH/POP` optimization
`b`: Disable `POP PULL` optimization
`c`: Disable Compare 0 optimizations
`d`: Disable load/store load/store optimization
`e`: Disable `LEA LEA` optimization
`f`: Disable load/store to `POP`/`PUSH` optimization
`g`: Disable load arithm store optimization
`h`: Disable `JSR`/`RTS` optimization
`i`: Disable `INC`/`DEC` Compare optimizations
`j`: Disable store store optimization
`k`: Disable `LEA 0` optimization
`l`: Disable `LEA` into addressing mode optimization
`m`: Disable `RET` optimization
`n`: Disable `BCLR`, `BCLR` Optimization)
`p`: Disable `PULL POP` optimization
`q`: Disable `PSHC PULC` optimization
`r`: Disable `BRA` to `RTS` optimization
`s`: Disable peephole 8-bit store combining
`t`: Disable `TFR TFR` optimization
`u`: Disable unused optimization
`x`: Disable peephole index optimization

Default

None

Defines

None

Pragmas

None

Description

If `-OnP` is specified, the whole peephole optimizer is disabled. To disable only a single peephole optimization, the optional syntax `-OnP=<char>` may be used, e.g., `-OnP=ef` disables `LEA/LEA` and `POP/PUSH` optimization. Refer to the Backend chapter for additional details.

Example

```
-OnP
```

See also

Peephole Optimizations

-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
```

-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(1) ...;`' into '...;'

-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=g

Disable compare size optimization

Examples (assume that `ch` is a character variable) :

Disable optimization of `((int)8 < (int)(unsigned char)ch)` into `(unsigned char) 8 < (unsigned char) ch;`

Disable optimization of `(int)(unsigned char)ch < (int)8` into `optimized to (unsigned char) ch < (unsigned char) 8;`

Disable optimization of `ch > (int)8` into `ch > (char)8;`

-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
```

-Or: Allocate Local Variables into Registers

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-Or`

Arguments

None

Default

None

Defines

`__OPTIMIZE_REG__`

Pragmas

None

Description

Allocate local variables (`char` or `int`) in registers. The number of local variables allocated in registers depends on the number of available registers. This option is useful when using variables as loop counters or switch selectors or if the processor requires register operands for multiple operations (e.g., RISC processors). Compiling with this option may increase your code size (spill and merge code).

NOTE This optimization may result in code that could be very hard to debug at the High-level Language level.

NOTE This optimization will not take effect for some backends.
For some backends the code does not change.

Example

```
-Or

int main(void) {
  int a, b;
  return a + b;
}
```

Case (1) without the -Or option (pseudo code):

```
tmp1 LD     a
tmp2 LD     b
tmp3 ADD    tmp1,tmp2
     RET    tmp3
```

Case (2) with the -Or option (pseudo code):

```
tmp1 ADD a,b
RET tmp1
```

-Ou and -Onu: Optimize Dead Assignments

Group

OPTIMIZATIONS

Scope

Function

Syntax

`-O(u|nu)`

Arguments

None

Default

Optimization enabled for functions containing no inline assembler code

Defines

None

Pragmas

None

Description

Optimize dead assignments. Assignments to local variables, not referenced later, are removed by the Compiler.

There are three possible settings for this option:

- `-Ou` is given

  Always optimize dead assignments (even if HLI is present in current function). Inline assembler accesses are not considered.

NOTE This option is not safe when accesses to local variables are contained in inline assembler code.

- `-Onu` is given

  The optimization does not take place. This generates the best possible debug information. The code is larger and slower than without -One.

- neither -Ou nor -Onu is given

  Optimize dead assignments if HLI is not present in the current function.

NOTE The compiler is not aware of `longjmp()` or `setjmp()` calls. These functions, those that are similar, may generate a control flow which is not recognized by the compiler. Therefore, be sure to either not use local variables in functions using `longjmp()` or `setjmp()` or switch this optimization off by using `-Onu`.

NOTE Dead assignments to volatile declared global objects are never optimized.

Example

```
-Ou

void main(int x) {
  f(x);
  x = 1;   /* this assignment is dead and is
              removed if -Ou is active */
}
```

-Pe: Preprocessing Escape Sequences in Strings

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
```

-PEDIV: Use EDIV instruction

Group

CODE GENERATION

Scope

Function

Syntax

`-PEDIV[={Div|Mod}]`

Arguments

`Div`: Use `EDIV` for divisions
`Mod`: Use `EDIV` for modulo instructions
Not specifying `Div` or `Mod`, `-PEDIV` means the same as specifying both after the assignment (`-PEDIV=DivMod`).

Default

None

Defines

None

Pragmas

None

Description

The HC12 instruction set contains an `EDIV` and an `EDIVS` instruction. Each instruction divides a 32-bit value by a 16-bit value giving a 16-bit quotient and a 16-bit remainder. The `EDIV` instruction handles the unsigned division case and the `EDIVS` the signed division case.

With this option enabled, the compiler generates an `EDIV` instructions instead of calling a division runtime routine for matching cases. When a 32-bit value is divided by a 16-bit value, only 16 bits of the result are used.

The `EDIV` instruction, as implemented in the HC12 hardware, does not calculate a result when an overflow occurs. When using `EDIV` to calculate `0x100001 % 0x10`, the EDIV instruction does not return `0x01` as a remainder because the quotient overflows. Therefore, the `EDIV` instruction cannot be used in a C-compatible code structure. When this option is enabled, the Compiler generates this instruction assuming that no overflow occurs. If an overflow occurs, the Compiler assumes it is insignificant.

Using this option may generate much faster and shorter code. But because this optimization is not ANSI-C compliant, this option must be enabled separately.

Examples

See Listing 5.55 through Listing 5.58 for examples of the PEDIV compiler option.

Listing 5.55 C source example

```
long Divisor;
int Dividend;
int Remainder;
void Div(void) {
  Remainder= Divisor%Quotient;
}
```

Listing 5.56 Div with -PEDIV generates the following disassembled code

```
    LDD   Divisor:0x2
    LDX   Dividend
    LDY   Divisor
    EDIVS
    STD   Remainder
```

Listing 5.57 Div without -PEDIV generates the following disassembled code

```
    LDD   Dividend
    JSR   _ILSEXT ; calls INT to LONG conversion routine
    PSHD
    PSHX
    LDD   Divisor:0x2
    LDX   Divisor
    JSR   _LMODS  ; calls the slow long division routine
    STD   Remainder
```

Listing 5.58 Example of usage

```
void main(void) {
  Divisor  = 0x12345678;
  Dividend = 0x4567;
  Div(); /* in these case both version work because */
         /* 0x12345678 / 0x4567 == 0x4326 <= 0x7FFF */
  Dividend = 0x10;
  Div(); /* here the function compiled with -PEDIV */
         /* does not return 8 in Remainder because */
         /* 0x12345678 / 0x10 == 0x1234567 > 0x7FFF */
}
```

-Pic: Generate Position-Independent Code (PIC)

Group

CODE

Scope

Function

Syntax

`-Pic`

Arguments

None

Default

None

Defines

__PIC__

Pragmas

None

Description

With this option enabled, the Compiler generates `PIC` (Position-independent Code). `PIC` is generated only for code (call of functions) and not for data. Instead of using `JSR` with extended (16-bit) addressing mode for function calls, the Compiler uses a PC-relative (`IDX2`) call. This ensures the branch distance is encoded instead of the absolute address (Listing 5.59).

Also, the Compiler uses an `LBRA` instead of a `JMP` for a local unconditional branch.

Listing 5.59 Function call using PIC

```
void foo(void);

void main(void) {
  foo(); // BSR foo instead of JSR foo
}
```

With -pic:
```
  0000 05fa0000      JMP   foo,PCR
```

Without -pic:
```
  0000 060000        JMP   foo
```

NOTE With `-Pic`, the code is larger and slower. Therefore, this should only be used whenever necessary.

Example

```
-Pic
```

See also

HC(S)12 Backend
-PicRTS: Call Runtime Support Position Independent compiler option
#pragma CODE_SEG: Code Segment Definition

-PicRTS: Call Runtime Support Position Independent

Group

CODE

Scope

Function

Syntax

`-PicRTS`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

When this option is enabled, the Compiler calls runtime functions independently in position-independent code position. This requires one additional byte per call and should only be done when the whole application, including the runtime support, must be position-independent. This option only affects position-independent functions. Runtime calls that are not position-independent functions are still done absolutely. This option is only useful when used together with:

- `#pragma CODE_SEG __PIC_SEG PicSegName` or with
- the `-Pic` option.

Example

```
-PicRTS
```

See also

Position-Independent Code (PIC)
#pragma CODE_SEG: Code Segment Definition
-Pic: Generate Position-Independent Code (PIC)

-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 Listing 5.60:

Listing 5.60 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 which do not contain macros of that nature.

Example

```
-Pio
```

-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
```

Use the compiler executable name instead of "`compiler`".

See also

Local Configuration File (usually project.ini)

-PSeg: Assume Objects are on Same Page

Group

CODE GENERATION

Scope

Function

Syntax

`-PSeg(All|NonDef|Obj)`

Arguments

None

Default

`NonDef`

Defines

None

Pragmas

None

Description

The compiler has to generate at least two accesses to access objects allocated in the `__far` area. First, the accessed page must be stored in the page register, and then the actual access takes place.

As an optimization, the compiler tries to avoid unnecessary page stores. If two memory accesses are using the same page, the second page store is avoided.

This option controls what the compiler assumes about the page of different objects:

- `-PSegAll`

  All objects in the same segment share the same page number. As a special case, all otherwise unallocated objects are in the same default segment.

- `-PSegNonDef`

  All objects in the same user-defined segment share the same page number. Objects in default segments do not share the same page number.

- `-PSegObj`

  Any two objects might have different page numbers. The compiler only optimizes page stores for the same object.

NOTE This option is effective only when directly accessing `__far` objects. It does not change accesses with a runtime routine.

Example

Consider the following example in the large memory model (Listing 5.61).

Listing 5.61 Example using the large memory model:

```
char i0_def_seg;
char i1_def_seg;
#pragma DATA_SEG __DPAGE_SEG DPAGE_CONTROLLED
char i2_user_seg;
char i3_user_seg;
void main(void) {
  i0_def_seg=56;
  i1_def_seg=56;
  i2_user_seg=56;
  i3_user_seg=56;
}
```

When compiled with the `-PSegAll` option, the `i0_def_seg` variable is on the same page as `i1_def_seg`, and the `i2_user_seg` variable is on the same page as the `i3_user_seg` variable. Therefore, the compiler sets the page register twice, as shown in Listing 5.62:

Listing 5.62 Listing 5.61 compiled with the -PSegAll option

```
Options : -CpDPAGE=0x34  -Ml -PSegAll

  0000 c638         LDAB  #56
  0002 8600         LDAA  #i0_def_seg:Page
  0004 5a34         STAA  52
  0006 7b0000       STAB  i0_def_seg
  0009 7b0000       STAB  i1_def_seg
  000c 8600         LDAA  #i2_user_seg:Page
  000e 5a34         STAA  52
  0010 7b0000       STAB  i2_user_seg
  0013 7b0000       STAB  i3_user_seg
  0016 0a           RTC
```

When compiled with the `-PSegNonDef` option, only the `i2_user_seg` variable is on the same page as the `i3_user_seg` variable. Therefore, the compiler sets the page register three times, as shown in Listing 5.63:

Listing 5.63 Listing 5.61 compiled with the -PSegNonDef option

```
Options : -CpDPAGE=0x34 -Ml -PSegNonDef

  0000 c638          LDAB  #56
  0002 8600          LDAA  #i0_def_seg:Page
  0004 5a34          STAA  52
  0006 7b0000        STAB  i0_def_seg
  0009 8600          LDAA  #i1_def_seg:Page
  000b 5a34          STAA  52
  000d 7b0000        STAB  i1_def_seg
  0010 8600          LDAA  #i2_user_seg:Page
  0012 5a34          STAA  52
  0014 7b0000        STAB  i2_user_seg
  0017 7b0000        STAB  i3_user_seg
  001a 0a            RTC
```

Finally, with the `-PSegObj` option, all variables may be on different pages. The page is set for every variable (Listing 5.64):

Listing 5.64 Listing 5.61 compiled with the -PSegObj option

```
Options: -CpDPAGE=0x34 -Ml -PSegObj

  0000 c638          LDAB  #56
  0002 8600          LDAA  #i0_def_seg:Page
  0004 5a34          STAA  52
  0006 7b0000        STAB  i0_def_seg
  0009 8600          LDAA  #i1_def_seg:Page
  000b 5a34          STAA  52
  000d 7b0000        STAB  i1_def_seg
  0010 8600          LDAA  #i2_user_seg:Page
  0012 5a34          STAA  52
  0014 7b0000        STAB  i2_user_seg
  0017 8600          LDAA  #i3_user_seg:Page
  0019 5a34          STAA  52
  001b 7b0000        STAB  i3_user_seg
  001e 0a            RTC
```

-Px4: Do Not Use ?BNE or ?BEQ

Group

CODE GENERATION

Scope

Function

Syntax

`-Px4`

Arguments

None

Default

None

Defines

`__PROCESSOR_X4__`

Pragmas

None

Description

Some processors do not support all HC12 instructions. The Compiler does not generate instructions and code patterns which do not work on all available processors when this option is used. The following points are affected by this option:

- None of the instructions below is generated:
  - `TBNE`
  - `TBEQ`
  - `IBNE`
  - `IBEQ`
  - `DBNE`
  - `DBEQ`
- Also, the overflow flag is not used after a `COM` instruction.

- With this option set, the inline assembler does not allow the use of the instructions listed in Item 1, above.
- The `__PROCESSOR_X4__ macro` is defined to allow different inline assembler code with conditional compilation.

Example

```
-Px4
```

-Qvtp: Qualifier for Virtual Table Pointers

Group

CODE GENERATION

Scope

Application

Syntax

`-Qvtp(none|far|near|rom|uni|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 It is useless to specify a qualifier which is not supported by the Backend (see Backend), 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).

-Rp (-Rpe, -Rpt): Large Return Value Type

Group

OPTIMIZATIONS

Scope

Application

Syntax

`-Rp(t|e)`

Arguments

`t`: Pass the large return value by pointer
`e`: Pass the large return value with temporary elimination

Default

`-Rpe`

Defines

None

Pragmas

None

Description

This option is supported by the Compiler 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 could not be passed in registers.

Consider the following source code in Listing 5.65:

Listing 5.65 Example source code

```
typedef struct { int i[10]; } S;

S F(void);
S s;
```

```
void main(void) {
  s = F();
}
```

In the above case, with `-Rpt`, the code will look like (Listing 5.66):

Listing 5.66 Pass large return value by pointer

```
void main(void) {
  S tmp;

  F(&tmp);
  s = tmp; /* struct copy */
}
```

The above approach is always correct but not efficient. The better solution is to pass the destination address directly to the callee making it unnecessary to declare a temporary and a struct copy in the caller (i.e., `-Rpe`), as shown below:

Listing 5.67 Pass large return value by temporary elimination

```
void main(void) {
  F(&s);
}
```

The above example may produce incorrect results for rare cases, e.g., if the F() function returns something which overlaps s. Because it is not possible for the Compiler to detect such rare cases, two options are provided: the `-Rpt` (always correct, but inefficient), or `-Rpe` (efficient) options.

-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 Listing 5.68 is predefined by the Compiler:

Listing 5.68 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_IEEE64__
__FLOAT_IS_DSP__

__DOUBLE_IS_IEEE32__
__DOUBLE_IS_IEEE64__
__DOUBLE_IS_DSP__

__LONG_DOUBLE_IS_IEEE32__
__LONG_DOUBLE_IS_IEEE64__
__LONG_DOUBLE_IS_DSP__

__LONG_LONG_DOUBLE_IS_IEEE32__
__LONG_LONG_DOUBLE_IS_IEEE64__
__LONG_LONG_DOUBLE_DSP__

__VTAB_DELTA_IS_8BIT__
__VTAB_DELTA_IS_16BIT__
__VTAB_DELTA_IS_32BIT__
__VTAB_DELTA_IS_64BIT__

__PTRMBR_OFFSET_IS_8BIT__
__PTRMBR_OFFSET_IS_16BIT__
__PTRMBR_OFFSET_IS_32BIT__
__PTRMBR_OFFSET_IS_64BIT__
```

Pragmas

None

Description

This option allows configurable type settings. The syntax of the option is:

```
-T{<type><format>}
```

For `<type>`, one of the keys listed in Table 5.9 may be specified:

Table 5.9 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. See Table 5.10.

Table 5.10 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 Table 5.11. 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. Refer to Sign of Plain Bitfields.

Table 5.11 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 Table 5.12 can be specified.

Table 5.12 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'
IEEE64 floating	'4'
DSP (32-bit)	'0'

Not all formats may be available for a target. See the Backend chapter 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 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 5.69 -Tsc1s2i2L4LL4f2d4Ld4LLd4e2 denotes:

```
signed char with 8 bits (sc1)
short and int with 16 bits (s2i2)
long, long long with 32 bits (L4LL4)
float with IEEE32 (f2)
double, long double and long long double with IEEE64 (d4Ld4LLd4)
enum with 16 bits (signed) (e2)
```

Listing 5.70 Restrictions

For integrity and compliance to ANSI, the following two rules have to be true:

```
sizeof(char)        <= sizeof(short)
sizeof(short)       <= sizeof(int)
sizeof(int)         <= sizeof(long)
sizeof(long)        <= sizeof(long long)
sizeof(float)       <= sizeof(double)
sizeof(double)      <= sizeof(long double)
sizeof(long double) <= sizeof(long long double)
```

NOTE 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`

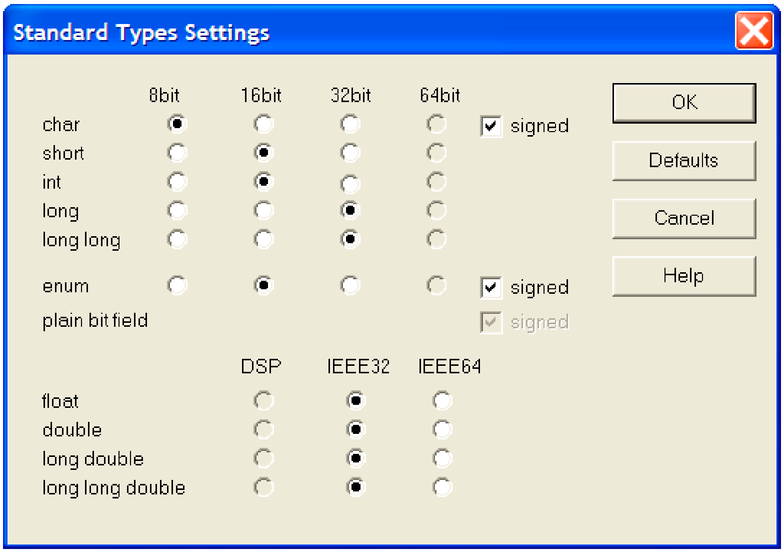
Setting char to 8 bits and unsigned, int to 32 bits and long long to 32 bits:
`-Tuc1i4LL4`

Setting float to IEEE32 and double to IEEE64:
`-Tf2d4`

The `-Tvtd` option allows you to change the delta value type inside virtual function tables. The default setting is 16-bit.

Another way to set this option is using the dialog box in the Graphical User Interface (Figure 5.4):

Figure 5.4 Standard Types Settings dialog box



	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	○	●	○

See also

Sign of Plain Bitfields

-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
```

-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 (e.g., Linker, Compiler, ...) is started 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: Display Notify Box option, a dialog box is still possible.

Example

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

-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

-Wmsg8x3: Cut filenames in Microsoft Format to 8.3

Group

MESSAGES

Scope

Compilation Unit

Syntax

```
-Wmsg8x3
```

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Some editors (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 Win95 or WinNT. 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

-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

-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

-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

-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

-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

-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 5.71).

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 Listing 5.71 for examples showing the differing message formats.

Listing 5.71 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

-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 Listing 5.72 for examples showing the differing message formats.

Listing 5.72 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

-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 Table 5.13 are supported (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

Table 5.13 Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	X:\Freescale\
%f	Path and name	X:\Freescale\mysourcefile
%n	filename	mysourcefile

Table 5.13 Message Format Specifiers (*continued*)

Format	Description	Example
%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, the path, or the extension contains a space	
%'	A ' if the filename, the path, or the extension contains a space	

Example

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

Produces a message in the following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

ERRORFILE: Error filename Specification
-WmsgFb (-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

-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 Table 5.14 are supported (assuming that the source file is `X:\Freescale\mysourcefile.cpph`):

Table 5.14 Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	X:\sources\
%f	Path and name	X:\sources\mysourcefile
%n	filename	mysourcefile

Table 5.14 Message Format Specifiers (*continued*)

Format	Description	Example
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space.	
%'	A ' if the filename, the path or the extension contains a space	

Example

```
-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

-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. Table 5.15 lists the supported formats.

Table 5.15 Message Format Specifiers

Format	Description	Example
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text

Table 5.15 Message Format Specifiers (*continued*)

Format	Description	Example
%%	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

-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. Table 5.16 lists the supported formats.

Table 5.16 Message Format Specifiers

Format	Description	Example
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text

Table 5.16 Message Format Specifiers (*continued*)

Format	Description	Example
%%	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

-WmsgNe: 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: Number of Information Messages
-WmsgNw: Number of Warning Messages

-WmsgNi: 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: Number of Error Messages
- -WmsgNw: Number of Warning Messages

-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

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 and so on).

e: Disables informal messages (e.g., memory model, floating point format, ...).

NOTE Depending on the application, the Compiler may not recognize all suboptions. In this case they are ignored for compatibility.

Example

```
-WmsgNu=c
```

-WmsgNw: 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: Number of Error Messages
- -WmsgNi: Number of Information Messagesi

-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

-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

-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

-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 Informationi

-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 an error listing file should be created. 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 ERRORFILE: Error filename Specification environment variable.

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

-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

Message C1801
-WmsgSe: Setting a Message to Error

-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

-W1: No 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: Number of Information Messages

-W2: No Information and 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: Number of Information Messages

-WmsgNw: Number of Warning Messages

6 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 Table 6.1.

Table 6.1 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.

NOTE If these macros do not have a value, the Compiler treats them as if they had been defined as shown: `#define __HIWARE__`

It is also possible to log all Compiler predefined defines to a file using the -Ldf: Log Predefined Defines to File compiler option.

Compiler Vendor Defines

Table 6.2 shows the defines identifying the Compiler vendor. Compilers in the USA may also be sold by ARCHIMEDES.

Table 6.2 Compiler Vendor Identification Defines

Name	Defined
`__HIWARE__`	always
`__MWERKS__`	always, set to 1

Product Defines

Table 6.3 shows the Defines identifying the Compiler. The Compiler is a HI-CROSS+ Compiler (V5.0.x).

Table 6.3 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

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 ‘Intel World’ uses Little Endian and the ‘Non-Intel World’ uses Big Endian.

The Compiler provides the “endian” macros listed in Table 6.4.

Table 6.4 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 difference between little and big endian (Listing 6.1).

Listing 6.1 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
```

Various Defines for Compiler Option Settings

The following table lists Defines for miscellaneous compiler option settings.

Table 6.5 Defines for Miscellaneous Compiler Option Settings

Name	Defined
__STDC__	-Ansi
__TRIGRAPHS__	-Ci
__CNI__	-Cni
__OPTIMIZE_FOR_TIME__	-Ot
__OPTIMIZE_FOR_SIZE__	-Os

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 6.2 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.

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.

Listing 6.3 show part of the contents of stdtypes.h (included from `stddef.h`).

Listing 6.3 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
```

Table 6.6 lists defines that deal with other possible implementations:

Table 6.6 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 'unsigned int'.
__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.

Macros for HC12

Table 6.7 shows the settings for the HC12 target:

Table 6.7 HC12 Compiler Defines

size_t Macro	Defined
__SIZE_T_IS_UCHAR__	never
__SIZE_T_IS_USHORT__	never
__SIZE_T_IS_UINT__	always
__SIZE_T_IS_ULONG__	never

Table 6.8 HC12 Compiler Pointer Difference Macros

ptrdiff_t Macro	Defined
__PTRDIFF_T_IS_CHAR__	never
__PTRDIFF_T_IS_SHORT__	never
__PTRDIFF_T_IS_INT__	always
__PTRDIFF_T_IS_LONG__	never

Division and Modulus

To ensure that the results of the "`/`" and "`%`" operators are defined correctly for signed arithmetic operations, both operands must be defined positive. (Refer to 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 Listing 6.4.

Listing 6.4 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.

Macros for HC12

Table 6.9 HC12 Compiler Modulo Operator Macros

Name	Defined
`__MODULO_IS_POSITIV__`	never

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. Table 6.10 lists these defines.

Table 6.10 Object-file Format Defines

Name	Defined
__HIWARE_OBJECT_FILE_FORMAT__	-Fh
__ELF_OBJECT_FILE_FORMAT__	-F1, -F2

Bitfield Defines

Bitfield Allocation

The Compiler provides six predefined macros to distinguish between the different allocations:

```
  __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 (Listing 6.5).

Listing 6.5 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 Listing 6.6:

Listing 6.6 __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.

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 (Listing 6.7):

Listing 6.7 Compatible optimized bitfields

```
struct{
  long b1:4;
  long b2:4;
} myBitfield;
31                     7  3  0
--------------------------------
|#######################|b2|b1|  -BfaTSRoff
--------------------------------
7      3     0
```

```
--------------
|  b2   |  b1 | -BfaTSRon
--------------
```

Sign of Plain Bitfields

For some architectures, the sign of a plain bitfield does not follow standard rules. Normally in the following (Listing 6.8):

Listing 6.8 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 (Listing 6.9):

Listing 6.9 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:

```
__PLAIN_BITFIELD_IS_SIGNED__    /* defined if plain bitfield
                                   is signed   */
__PLAIN_BITFIELD_IS_UNSIGNED__  /* defined if plain bitfield
                                   is unsigned */
```

Macros for HC12

Table 6.11 identifies the implementation in the Backend.

Table 6.11 HC12 Compiler—Backend Macro

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

Type Information Defines

The Flexible Type Management sets the defines to identify the type sizes. Table 6.12 lists these defines.

Table 6.12 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
__CHAR_IS_32BIT__	see -T option or Backend
__CHAR_IS_64BIT__	see -T option or Backend
__SHORT_IS_8BIT__	see -T option or Backend

Table 6.12 Type Information Defines (*continued*)

Name	Defined
__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_IEEE64__	see -T option or Backend

Table 6.12 Type Information Defines (*continued*)

Name	Defined
__FLOAT_IS_DSP__	see -T option or Backend
__DOUBLE_IS_IEEE32__	see -T option or Backend
__DOUBLE_IS_IEEE64__	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_IEEE64__	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_IEEE64__	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

7

Compiler Pragmas

A pragma (Listing 7.1) 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 7.1 The syntax of a pragma

```
#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.

Pragma Details

This section describes each Compiler-available pragma. The pragmas are listed in alphabetical order and are divided into separate tables. Table 7.1 lists and defines the topics that appear in the description of each pragma.

Table 7.1 Pragma documentation topics

Topic	Description
Scope	Scope of pragma where it is valid. (See Table 7.2, 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.

Table 7.2 is a description of the different scopes for pragmas.

Table 7.2 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.

#pragma align (on|off): Turn alignment on or off

Scope

Until the next `align` pragma

Syntax

```
#pragma align  (on|off)
```

Synonym

None.

Arguments

on: the HCS12X compiler uses the same alignment as the XGATE compiler
off: the HCS12X compiler uses no alignment

Default

```
#pragma align off
```

Description

The pragma align simplifies the sharing of variables between the HCS12X and the XGATE cores. The HCS12X core does not need any alignment. However, if some data structures are accessed from both the HCS12X and the XGATE, their layouts must be identical. This pragma causes the HCS12X compiler to insert the same alignment bytes as the XGATE compiler. Therefore, enabling it causes potentially larger data structures.

NOTE This pragma does not ensure that the same data size or encoding is used for the data representation. The HCS12X supports 3-byte pointers and 8-byte doubles. However, the XGATE always allocates pointers as two bytes and doubles as four bytes.
Also note that the different cores are using a different encoding for pointers.

Example

```
#pragma align on
struct {
  char ch;        /* offset: 0 */
  int i;          /* offset: 2 */
} s_aligned;

#pragma align off
struct {
  char ch;        /* offset: 0 */
  int i;          /* offset: 1 */
} s;
```

#pragma CODE_SEG: Code Segment Definition

Scope

Until the next `CODE_SEG` pragma

Syntax

```
#pragma CODE_SEG (<Modif>  <Name>|DEFAULT)
```

Synonym

`CODE_SECTION`

Arguments

Listing 7.2 Some of the following strings may be used for <Motif>:

```
__DIRECT_SEG  (compatibility alias: DIRECT)
__NEAR_SEG    (compatibility alias:  NEAR)
__CODE_SEG    (compatibility alias:  CODE)
__FAR_SEG     (compatibility alias:  FAR)
__DPAGE_SEG   (compatibility alias:  DPAGE)
__EPAGE_SEG   (compatibility alias:  EPAGE)
__PPAGE_SEG   (compatibility alias:  PPAGE)
__RPAGE_SEG   (compatibility alias:  RPAGE)
__GPAGE_SEG   (compatibility alias:  GPAGE)
__PIC_SEG     (compatibility alias:  PIC)
```

NOTE The compatibility alias should not be used 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.

The meaning of these segment modifiers are backend-dependent. Refer to the HC(S)12 Backend chapter for information on supported modifiers and their definitions.

`<Name>`: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the `PLACEMENT` section. Refer to the Linker section of the Build Tools manual for details.

Default

`DEFAULT`

Description

This pragma specifies where the function segment it is allocated. The segment modifiers also specify the function's calling convention. The `CODE_SEG` pragma sets the current code segment. This segment places all new function definitions. Also, all function declarations get the current code segment when they occur. The segment modifiers of this segment determine the calling convention.

The `CODE_SEG` pragma affects function declarations as well as definitions. Ensure that all function declarations and their definitions are in the same segment.

The synonym `CODE_SECTION` has exactly the same meaning as `CODE_SEG`.

Listing 7.3 shows program code segments allocated with `CODE_SEG` pragmas.

Listing 7.3 CODE_SEG examples

```
/* in a header file */
#pragma CODE_SEG __FAR_SEG MY_CODE1
extern void f(void);
#pragma CODE_SEG MY_CODE2
extern void h(void);
#pragma CODE_SEG DEFAULT

/* in its corresponding C file: */
#pragma CODE_SEG __FAR_SEG MY_CODE1
void f(void){ /* f has FAR calling convention */
  h(); /* calls h with default calling convention */
}
#pragma CODE_SEG MY_CODE2
void h(void){ /* f has default calling convention */
  f(); /* calls f() with the FAR calling convention */
}
#pragma CODE_SEG DEFAULT
```

NOTE Not all backends support a `FAR` calling convention.

NOTE The calling convention can also be specified with a supported keyword. The default calling convention is chosen with the memory model.

Listing 7.4 has some examples of improper `CODE_SEG` pragma usage.

Listing 7.4 Improper pragma usage

```
#pragma DATA_SEG DATA1
#pragma CODE_SEG DATA1
/* error: same segment name has different types! */

#pragma CODE_SEG DATA1
#pragma CODE_SEG __FAR_SEG DATA1
/* error: same segment name has modifiers! */

#pragma CODE_SEG DATA1
void g(void);
#pragma CODE_SEG DEFAULT
void g(void) {}
/* error: g() is declared in two different segments */
#pragma CODE_SEG __FAR_SEG DEFAULT
/* error: modifiers for the DEFAULT segment are not allowed */
```

See also

HC(S)12 Backend chapter
Segmentation
Linker section of the Build Tools manual
#pragma CONST_SEG: Constant Data Segment Definition
#pragma DATA_SEG: Data Segment Definition
#pragma STRING_SEG: String Segment Definition
#pragma STRING_SEG: String Segment Definition compiler option

#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 7.5 Some of the following strings may be used for <Modif>:

```
__SHORT_SEG   (compatibility alias: SHORT)
__DIRECT_SEG  (compatibility alias: DIRECT)
__NEAR_SEG    (compatibility alias: NEAR)
__CODE_SEG    (compatibility alias: CODE)
__FAR_SEG     (compatibility alias: FAR)
__DPAGE_SEG   (compatibility alias: DPAGE)
__EPAGE_SEG   (compatibility alias: EPAGE)
__PPAGE_SEG   (compatibility alias: PPAGE)
__RPAGE_SEG   (compatibility alias: RPAGE)
__GPAGE_SEG   (compatibility alias: GPAGE)
```

NOTE A compatibility alias should not be used 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. Refer to the HC(S)12 Backend chapter to find the supported modifiers and their meanings. The `__SHORT_SEG` modifier specifies a segment which is accessed with 8-bit addresses.

`<Name>`: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker section of the Build Tools manual for details.

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

Constants are allocated in the current data segment that is defined with the #pragma DATA_SEG: Data Segment Definition in the HIWARE object-file format when the -Cc: Allocate Constant Objects into ROM compiler option is not specified and until the first #pragma CONST_SEG occurs in the source. 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.

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

Listing 7.6 shows code that uses the CONST_SEG pragma.

Listing 7.6 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
```

Listing 7.7 shows code that uses the `CONST_SEG` pragma *improperly*.

Listing 7.7 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

HC(S)12 Backend chapter
Segmentation
Linker section of the Build Tools manual
#pragma CODE_SEG: Code Segment Definition
#pragma DATA_SEG: Data Segment Definition
#pragma STRING_SEG: String Segment Definition
#pragma INTO_ROM: Put Next Variable Definition into ROM
-Cc: Allocate Constant Objects into ROM compiler option

#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 only generated when the `-La` compiler option is specified together with a header file containing `#pragma CREATE_ASM_LISTING ON`.

Listing 7.8 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)

#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 7.9 Some of the following strings may be used for <Motif>:

```
__SHORT_SEG   (compatibility alias: SHORT)
__DIRECT_SEG  (compatibility alias: DIRECT)
__NEAR_SEG    (compatibility alias: NEAR)
__CODE_SEG    (compatibility alias: CODE)
__FAR_SEG     (compatibility alias: FAR)
__DPAGE_SEG   (compatibility alias: DPAGE)
__EPAGE_SEG   (compatibility alias: EPAGE)
__PPAGE_SEG   (compatibility alias: PPAGE)
__RPAGE_SEG   (compatibility alias: RPAGE)
__GPAGE_SEG   (compatibility alias: GPAGE)
```

NOTE A compatibility alias should not be used 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.

The `__SHORT_SEG` modifier specifies a segment which is accessed with 8-bit addresses. The meaning of these segment modifiers are backend-dependent. Read the backend chapter to find the supported modifiers and their meanings.

`<Name>`: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the `PLACEMENT` part. Please refer to the linker manual for details.

Default

DEFAULT

Description

This pragma allocates variables into a segment. This segment is then located in the link parameter file to specific addresses.
The DATA_SEG pragma sets the current data segment. This segment is used to place all variable declarations. The default segment is set with:

`#pragma DATA_SEG DEFAULT`

Constants are also allocated in the current data segment in the HIWARE object-file format when the option -cc is not specified and no “#pragma CONST_SEG” occurred in the source. When using the -Cc: Allocate Constant Objects into ROM compiler option and the ELF object-file format, constants are not allocated in the data segment.

The DATA_SEG pragma also affects data declarations, as well as definitions. Ensure that all variable declarations and definitions are in the same segment.

Some compiler optimizations assume that objects having the same segment are together. Backends supporting banked data, for example, may set the page register only once if two accesses two different variables in the same segment are done. This is also the case for the DEFAULT segment. When using a paged access to constant variables, put one segment on one page in the link parameter file.

When #pragma INTO_ROM: Put Next Variable Definition into ROM is active, the current data segment is not used.

The DATA_SECTION synonym has exactly the same meaning as DATA_SEG.

Example

Listing 7.10 shows source code that uses the DATA_SEG pragma.

Listing 7.10 Using the DATA_SEG pragma

```
/* in a header file */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
extern int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
extern int j_custom;
#pragma DATA_SEG DEFAULT

/* in the corresponding C file : */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
int j_custom;
```

```
#pragma DATA_SEG DEFAULT

void main(void) {
  i = 1; /* may use short access */
  j = 5;
}
```

Listing 7.11 shows code that uses the `DATA_SEG` pragma *improperly*.

Listing 7.11 Improper use of the DATA_SEG pragma

```
#pragma DATA_SEG DATA1
#pragma CONST_SEG DATA1 /* error: segment name has different types! */

#pragma DATA_SEG DATA1
#pragma DATA_SEG __SHORT_SEG DATA1
/* error: segment name has modifiers! */

#pragma DATA_SEG DATA1
extern int i;
#pragma DATA_SEG DEFAULT
int i; /* error: i is declared in different segments */

#pragma DATA_SEG __SHORT_SEG DEFAULT
/* error: modifiers for the DEFAULT segment are not allowed */
```

See also

HC(S)12 Backend chapter
Segmentation
Linker section of the Build Tools manual
#pragma CODE_SEG: Code Segment Definition
#pragma CONST_SEG: Constant Data Segment Definition
#pragma STRING_SEG: String Segment Definition
#pragma INTO_ROM: Put Next Variable Definition into ROM
-Cc: Allocate Constant Objects into ROM compiler option

#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 7.12 Using an INLINE pragma to inline a function

```
int i;
#pragma INLINE
static void foo(void) {
  i = 12;
}
void main(void) {
  foo(); // results in inlining ‘i = 12;’
}
```

See also

#pragma NO_INLINE: Do not Inline next function definition
-Oi: Inlining compiler option

#pragma INTO_ROM: Put Next Variable Definition into ROM

Scope

Data Definition

Syntax

```
#pragma INTO_ROM
```

Synonym

None

Arguments

None

Default

None

Description

This pragma forces the next (non-constant) variable definition to be `const` (together with the `-Cc` compiler option).

The pragma is active only for the next single variable definition. A following segment pragma (`CONST_SEG`, `DATA_SEG`, `CODE_SEG`) disables the pragma.

NOTE This pragma is only useful for the HIWARE object-file format (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

Listing 7.13 presents some examples which use the `INTO_ROM` pragma.

Listing 7.13 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 Constant Objects into ROM compiler option

#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 do have no meaning other than one name can only contain one content. 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` in different object files and issues an message if a different `CONTENT` is given for different object files.

NOTE This pragma only works with the ELF object-file format.

Example

Apart from extended functionality implemented in the Linker or Debugger, this feature can also be used for user-defined link-time consistency checks:

Using the code shown in Listing 7.14 in a header file used by all compilation units, the Linker will issue a message if the object files built with `_DEBUG` are linked with object files built without it.

Listing 7.14 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
```

#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 if the `-Cu` option is set for the following single function.

Listing 7.15 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

#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 Listing 7.16 the pragma accesses declarations and definitions.

Listing 7.16 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++;
}
```

#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 Listing 7.17, parentheses ( ) were omitted.

Listing 7.17 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

#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` is generated independently of this pragma. The C code may therefore not work since it could access variables on the stack which were not allocated.

This pragma is only 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 HLI only functions should use a `#pragma NO_ENTRY` and a `#pragma NO_EXIT` to avoid generation of any additional frame instructions by the compiler.

The code generated in a function with `#pragma NO_ENTRY` may not be safe. It is assumed that the user ensures stack use.

WARNING! Not all backends support this pragma. Some may still generate entry code even if this pragma is specified.

Example

Listing 7.18 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 7.18 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

#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 HLI only functions should use a `#pragma NO_ENTRY` and a `#pragma NO_EXIT` 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 stack usage.

NOTE Not all backends support this pragma. Some may still generate exit code even if this pragma is specified.

Example

Listing 7.19 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 7.19 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

#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 not be safe. It is assumed that the user ensures stack usage.

NOTE Not all backends support this pragma. Some may still generate frame code even if this pragma is specified.

Example

Listing 7.20 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 7.20 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

#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 to inline the next function in the source. The pragma is used to avoid to inline a function which would be otherwise inlined because of the `-Oi` compiler option.

Listing 7.21 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 compiler option

#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 7.22 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 compiler option

#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 Listing 7.23 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 7.23 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

#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 7.24 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

#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 compiler option

#pragma OPTION: Additional Options

Scope

`Compilation Unit or until the next OPTION pragma`

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 Listing 7.25 shows how to compile only a single function with the additional `-Or` option.

Listing 7.25 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 Listing 7.26 show *improper* uses of the `OPTION` pragma.

Listing 7.26 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
```

#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 RALLOC_OBJ` to tell the compiler how to change calling conventions for each object. See the Linker manual for details.

Example

Listing 7.27 uses the `REALLOC_OBJ` pragma to reallocate the `evaluate.o` object file.

Listing 7.27 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

#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 7.28 Some of the following strings may be used for <Motif>:

```
__DIRECT_SEG  (compatibility alias:  DIRECT)
__NEAR_SEG    (compatibility alias:  NEAR)
__CODE_SEG    (compatibility alias:  CODE)
__FAR_SEG     (compatibility alias:  FAR)
__DPAGE_SEG   (compatibility alias:  DPAGE)
__EPAGE_SEG   (compatibility alias:  EPAGE)
__PPAGE_SEG   (compatibility alias:  PPAGE)
__RPAGE_SEG   (compatibility alias:  RPAGE)
__GPAGE_SEG   (compatibility alias:  GPAGE)
```

NOTE A compatibility alias should not be used 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. Read the backend chapter to find the supported modifiers and their definitions.

`<Name>`: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the `PLACEMENT` part. Please refer to the linker manual for details.

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

Listing 7.29 is an example of the `STRING_SEG` pragma allocating strings into a segment with the name, `STRING_MEMORY`.

Listing 7.29 Using a STRING_SEG pragma to allocate a segment for strings

```
#pragma STRING_SEG STRING_MEMORY
char* p="String1";
void f(char*);
void main(void) {
  f("String2");
}
#pragma STRING_SEG DEFAULT
```

See also

HC(S)12 Backend
Segmentation
Linker section of the Build Tools manual
#pragma CODE_SEG: Code Segment Definition
#pragma CONST_SEG: Constant Data Segment Definition
#pragma DATA_SEG: Data Segment Definition

#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

Listing 7.30 and Listing 7.31 present two examples of the `TEST_CODE` pragma.

Listing 7.30 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 (Listing 7.31).

Listing 7.31 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

#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 HC(S)12 Backend chapter for details.

Example

Listing 7.32 marks the `MyInterrupt()` function as an interrupt function.

Listing 7.32 Using the TRAP_PROC pragma to mark an interrupt function

```
#pragma TRAP_PROC
void MyInterrupt(void) {
  ...
}
```

See also

interrupt keyword

8

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.

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.

Keywords

See Listing 8.1 for the complete list of ANCSI-C keywords.

Listing 8.1 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
```

Preprocessor Directives

The Compiler supports the full set of preprocessor directives as required by the ANSI standard (Listing 8.2.

Listing 8.2 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.

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):

Pointer Qualifiers

Pointer qualifiers (Listing 8.3) 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 8.3 Pointer qualifiers

```
__far  (alias far)
__near (alias near)__dptr
__eptr
__pptr
```

```
__rptr
```

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 8.4).

Listing 8.4 far and near can be defined in the hidef.h file

```
#define far   /* no far keyword supported  */
#define near  /* no near keyword supported */
```

Special Keywords

ANSI-C was not designed with embedded controllers in mind. The listed keywords (Listing 8.5) do not conform to ANSI standards. However, they do enable an easy way to achieve good results from code used for embedded applications.

Listing 8.5 Special (non-ANSI) keywords

```
     __alignof__
     __va_sizeof__
     __interrupt (alias interrupt)
     __asm (aliases _asm and asm)
```

NOTE *See section Non-ANSI Keywords in the HC(S)12 Backend for more details.*

You can use the `__interrupt` keyword to mark functions as interrupt functions, and to link the function to a specified interrupt vector number (not supported by all backends).

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 0s or 1s (Listing 8.6).

Listing 8.6 Demonstration of a binary constant

```
#define myBinaryConst 0b01011
  int i;
```

```
void main(void) {
  i = myBinaryConst;
}
```

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.

#warning directive

The #warning directive (Listing 8.7) is used as it is similar to the #error directive.

Listing 8.7 #warning directive.

```
#ifndef MY_MACRO

  #warning "MY_MACRO set to default"
  #define MY_MACRO 1234
#endif
```

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 (Listing 8.8).

Listing 8.8 Assigning global variables to specific addresses

```
#pragma DATA_SEG [__SHORT_SEG] <segment_name>
```

setting the `PLACEMENT` section in the Linker parameter file. An older method of accomplishing this is shown in Listing 8.9.

Listing 8.9 Another means of assigning global variables to specific addresses

```
<segment_name> INTO  READ_ONLY <Address> ;
```

Listing 8.10 is an example using correctly and incorrectly the global variable address modifier and Listing 8.11 is a possible PRM file that corresponds with the example Listing.

Listing 8.10 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 8.11 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
```

Variable Allocation using @"SegmentName"

Sometimes it is useful to have the variable directly allocated in a named segment instead of using a `#pragma`. Listing 8.12 is an example of how to do this.

Listing 8.12 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. It should be defined 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 8.13).

Listing 8.13 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 8.14 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
```

Absolute Functions

Sometimes it is useful to call a absolute function (e.g., a special function in ROM). Listing 8.15 is a simple example of how this could be done using normal ANSI-C.

Listing 8.15 Absolute function

```
#define erase ((void(*)(void))(0xfc06))
void main(void) {
  erase(); /* call function at address 0xfc06 */
}
```

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, the absolute variable is not linked in HIWARE format by default. Instead, 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 Listing 8.16 shows how the linker handles different absolute variables.

Listing 8.16 Linker handling of absolute variables

```
      char i;         /* zero out            */
      char j = 1;     /* zero out, copy-down */
const char k = 2;     /* download            */
      char I@0x10;    /* no zero out!        */
      char J@0x11 = 1;/* copy down           */
const char K@0x12 = 2;/* HIWARE: copy down / ELF: download! */
static       char L@0x13;      /* no zero out! */
static       char M@0x14 = 3;  /* copy down    */
static const char N@0x15 = 4;  /* HIWARE: copy down, ELF: download */

void interrupt 2 MyISRfct(void) {} /* download, always linked! */
  /* vector number two is downloaded with &MyISRfct */

void 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.

The __far Keyword

The keyword far is a synonym for __far, which is not allowed when the -Ansi: Strict ANSI compiler option is present.

NOTE *See the Non-ANSI Keywords section in the HC12 Backend* chapter.

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.

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.

__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);
```

__far and typedef Names

If the array type has been defined 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`".

__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, Listing 8.17 and Listing 8.18 clarify this behavior:

Listing 8.17 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 8.18 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 access to the variable (addressing mode) and NOT the allocation.*

__far and C++ Classes

If a member function gets the modifier `__far`, the “`this`” pointer is a `__far` pointer. This is useful, if for instance, if the owner class of the function is not allocated on the default data page. See Listing 8.19.

Listing 8.19 __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
}
```

__far and C++ References

The `__far` modifier is applied to references. This is useful if it is a reference to an object outside of the default data page. For example:

```
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;
};
```

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. See the Backend chapter for details.

Table 8.1 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

Table 8.1 Interpretation of the __far Keyword (*continued*)

Declaration	Allowed	Type Description
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

__near Keyword

NOTE *See the Non-ANSI Keywords section in the HC(S)12 Backend.*

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 used instead of the `__far` keyword. It is used in situations where non qualified pointers are __far and an explicit `__near` access should be specified or where the __near calling convention must be explicitly specified.

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 8.2 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

Table 8.2 Interpretation of the __near Keyword (*continued*)

Declaration	Allowed	Type Description
int __far *i;	OK	pointer to a __far object
int * __near i;	OK	__near 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 __near pointer to void

__dptr, __eptr, __pptr, __rptr pointer qualifier keywords (HCS12X only)

The pointer qualifiers `__dptr`, `__eptr`, `__pptr`, and `__rptr` specify which page register should be used for a certain pointer type. With this information, more efficient code can be generated to perform the actual access.

NOTE *These pointer qualifiers are only supported for code generated for the HCS12X.*
The `__dptr` is only provided for symmetry reasons as the HCS12X does not actually have a DPAGE register.

Table 8.3 HCS12X only pointer qualifiers (not for HC12/HCS12)

qualifier	Address Kind	Description
__far	global	__far pointers use the HCS12X G-Load and G-Store instructions. Therefore __far pointers can point to any address of the HCS12X Note: as __far pointers use global addressing and the other pointer types use logical addressing, assignments to or from far pointers to other pointer types is relatively expensive Note: For the HC12/HCS12 __far pointers are also supported but have a different semantic.
__dptr	logical	not actually used as only a HC12 A4 does support a DPAGE register
__eptr	logical	Pointer to the paged EEProm area of a HCS12X
__pptr	logical	Pointer to the paged Flash area of a HCS12X Note: __pptr pointers directly set the page register. Therefore code using __pptr pointers must not be in the banked area. If code in the banked area should access another page, use a __far pointer instead.
__rptr	logical	Pointer to the paged RAM area of a HCS12X

Compatibility

`__far` pointers and normal pointers 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).

__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 (Listing 8.20):

Listing 8.20 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]))
```

__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 8.21).

Listing 8.21 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 8.22).

Listing 8.22 __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. They need a different space on the open parameter calls stack for some processors.*

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 the HS(S)12 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 (Listing 8.23). It specifies a function to be an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that should contain 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 8.23 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
```

__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 8.24).

See the inline assembler section in the backend chapter for details.

Listing 8.24 Examples of the __asm keyword

```
  __asm {
    nop
    nop ; comment
  }
  asm ("nop; nop");
  __asm("nop\n nop");
  __asm "nop";
  __asm nop;
#asm
  nop
  nop
#endasm
```

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.

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.

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 Listing 8.25).

Listing 8.25 Initialization using a non constant

```
void main() {
  struct A {
    struct A *n;
  } v={&v}; /* the address of v is not constant */
}
```

Sign of char

The ANSI-C standard leaves it open, whether the data type `char` is signed or unsigned. Check the Backend chapter for data about default settings.

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.*

Translation Limitations

This section describes the internal limitations of the Compiler. Some limitations are stack limitations depending 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 (e.g., Windows NT), or if it is a 16-bit Compiler running on a 16-bit platform (e.g., Windows for Workgroups).

The ANSI-C column in Table 8.4 below shows 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 8.4 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 8.4 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 8.5 Translation Limitations (non-ANSI)

Limitation	Description
Type Declarations	Derived types must not contain more than 100 components.
Labels	There may be at most 16 other labels within one procedure.
Macro Expansion	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.
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.

ANSI-C Standard

This section provides a short overview about the implementation (see also ANSI Standard 6.2) of the ANSI-C conversion rules.

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.

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.

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`.

Order of Operand Evaluation

The priority order of operators and their associativity is listed in Listing 8.26.

Listing 8.26 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. Listing 8.27 has some examples of operator precedence

Listing 8.27 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!
```

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 8.28 Size relationships among the integer types

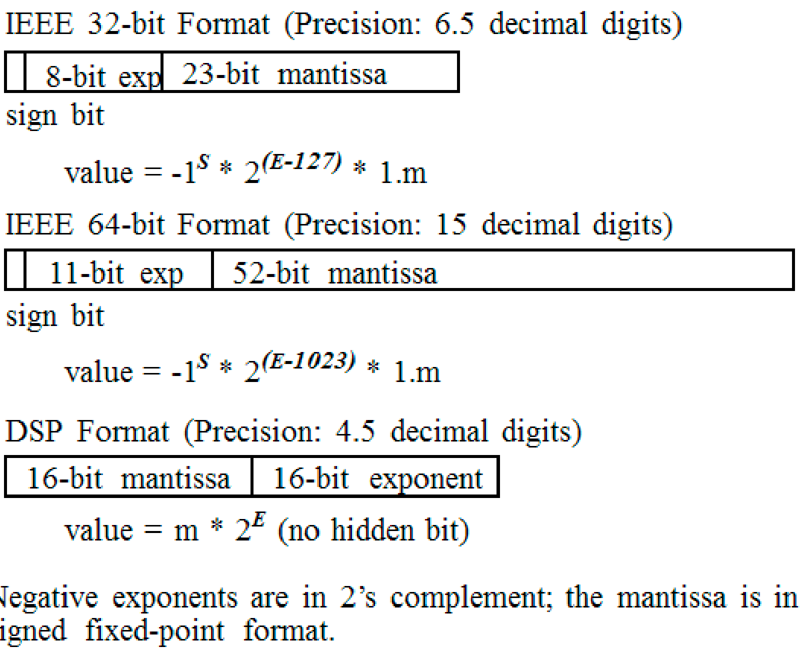
```
sizeof(char)   <= sizeof(short)
sizeof(short)  <= sizeof(int)
sizeof(int)    <= sizeof(long)
sizeof(long)   <= sizeof(long long)
sizeof(float)  <= sizeof(double)
sizeof(double)<= sizeof(long double)
```

Floating-Type Formats

The Compiler supports two IEEE floating point formats: IEEE32 and IEEE64. There may also be a DSP format supported by the processor. Figure 8.1 shows these three formats.

Floats are implemented as IEEE32, and doubles as IEEE64. This may vary for a specific Backend, or possibly, both formats may not be supported. Please check the Backend chapter for details, default settings and supported formats.

Figure 8.1 Floating-point formats


IEEE 32-bit Format (Precision: 6.5 decimal digits)
8-bit exp
23-bit mantissa
sign bit
value = $-1^S * 2^{(E-127)} * 1.m$
IEEE 64-bit Format (Precision: 15 decimal digits)
11-bit exp
52-bit mantissa
sign bit
value = $-1^S * 2^{(E-1023)} * 1.m$
DSP Format (Precision: 4.5 decimal digits)
16-bit mantissa
16-bit exponent
value = $m * 2^E$ (no hidden bit)

Negative exponents are in 2's complement; the mantissa is in signed fixed-point format.

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)

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 Listing 8.29:

Listing 8.29 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 Listing 8.30.

Listing 8.30 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)
```

Representation of 500.0 in IEEE64 Format

The exponent in IEEE64 has a fixed offset of 1023 to always have positive values:

```
exp (500.0,IEEE64) = 8 + 1023 == 407 (hex) == 10000000111
                                                  (bin)
```

The IEEE64 format is similar to IEEE32 except that more bits are available to represent the exponent and the mantissa. The IEEE64 representation of decimal 500 is shown in Listing 8.31.

Listing 8.31 Representation of decimal 500.0 in IEEE64

```
500.0 (dec) =
  0 (sign) 10000000111 (exponent)
  1111010000000000000000000000000000000000000000000000 (mantissa)
      (IEEE64 as bin)
  0100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000
  0000 0000 0000 0000 0000 (IEEE64 as bin)
  40 7f 40 00 00 00 00 00 (IEEE64 as hex)
```

The IEEE64 representation of decimal -500 is shown in Listing 8.32.

Listing 8.32 Representation of decimal -500.0 in IEEE64

```
-500.0 (dec) =
  1 (sign) 10000000111 (exponent)
  1111010000000000000000000000000000000000000000000000 (mantissa)
      (IEEE64 as bin)
  1100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000
  0000 0000 0000 0000 0000 (IEEE64 as bin)
= c0 7f 40 00 00 00 00 00 (IEEE64 as hex)
```

NOTE *The IEEE formats recognize several special bit patterns for special values. The number 0 (zero) is encoded by the bit pattern consisting of zero bits only. Other special values such as "Not a number", "infinity", -0 (minus zero) and denormalized numbers do exist. Please refer to the IEEE standard documentation for details.*
Except for the 0 (zero) and -0 (minus zero) special formats, not all special formats may be supported for specific backends.

Representation of 500.0 in DSP Format

Convert 500.0 from the decimal representation to a representation with base 2. In contradiction to IEEE, DSP normalizes the mantissa between 0 and 1 and not between 1 and 2. This makes it possible to also represent 0, which must have a special pattern in IEEE. Also, the exponent is different from IEEE.

`value = (-1)^s * m*2^e`
where sign is 1 or -1,
    `1 > m >= 0, and`
    exp is an integral number.

For 500 this gives:

- sign (500.0) = 1
- mant (500.0,DSP) = 0.9765625
- exp (500.0,DSP) = 9

Next convert the mantissa into its binary representation (Listing 8.33).

Listing 8.33 Representation of 500 in DSP format

```
mant (500.0, DSP) = 0.9765625 (dec)
  = 0*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)
            + 1*2^(-5) + 0*2^(-6) + 1*2^(-7) + 0*....
  = 0.1111101000... (bin).
```

Because this number is computed to be always larger or equal to 0 and smaller than 1, there is always a 0 in front of the decimal point. For the remaining steps this constant is left out to save space. There is always a 1 after the decimal point, except for 0 and intermediate results. This bit is encoded, so the DSP looses one additional bit of precision compared with IEEE.

```
mant (500.0, DSP, cut) = .1111101000... .
```

The exponent must also be converted to binary format:

```
exp (500.0,DSP) = 9 == 09 (hex) ==  1001 (bin)
```

Negative exponents are encoded by the 2’s representation of the positive value.

The sign is encoded into the mantissa by taking the 2’s complement for negative numbers and adding a 1 bit in the front. For DSP and positive numbers a 0 bit is added at the front.

```
mant(500.0, DSP) = 0111110100000000 (bin)
```

The twos complement is taken for negative numbers:

```
mant(-500.0, DSP) = 1000001100000000 (bin)
```

Finally the mantissa and the exponent must be joined according to Figure 8.1:

The DSP representation of decimal 500 is shown in Listing 8.34.

Listing 8.34 Representation of decimal 500.0 in DSP

```
500.0 (dec)
= 7D 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 7D 00 00 09 (DSP as hex)
= 0111 1101 0000 0000 0000 0000 0000 1001 (DSP as bin)
```

The DSP representation of decimal -500 is shown in Listing 8.35.

Listing 8.35 Representation of decimal -500.0 in DSP

```
-500.0 (dec)
= 83 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 83 00 00 09 (DSP as hex)
= 1000 0011 0000 0000 0000 0000 0000 1001 (DSP as bin)
```

NOTE *The order of the byte representation of a floating point value depends on the byte ordering of the backend. The first byte in the previous diagrams must be considered as the most significant byte.*

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 Listing 8.36 for one reason to use a volatile declaration.

Listing 8.36 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!
  }
}
```

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 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"*, 2^ND^ ed., the use of bitfields is equivalent to using bit masks to which the operators &, |, ~, |= or &= are applied. In fact, the Compiler translates bitfield operations to bit mask operations.

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 (Listing 8.37).

Listing 8.37 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)` does not make sense, i.e., it 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 8.38).

Listing 8.38 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.

Recommendations

In order to save memory, it recommended to implement globally accessible boolean flags as unsigned bitfields of width 1. However, it is not recommend using bitfields for other purposes because:

- Using bitfields to describe a bit pattern in memory is not portable between Compilers, even on the same target, as different Compilers may allocate bitfields differently.

For information about how the Compiler allocates bitfields, see the Data Types section in the HC(S)12 Backend chapter.

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 8.39 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 HIWARE object file format, constants are put into the DATA_SEG if no CONST_SEG was specified. 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. Please refer to the backend chapter for data about which modifiers are supported.

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.

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) */
```

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)     */
```

Optimizations

The Compiler applies a variety of code-improving techniques under the term "`optimization`". This section provides a short overview about the most important optimizations.

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.

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, the C source code should already contain a shift, or the value to be shifted should be unsigned.*

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.

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.

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.

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 Listing 8.40). This results in using less memory space.

Listing 8.40 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 Listing 8.41).

Listing 8.41 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;
}
```

Tree Rewriting

The structure of the intermediate code between Frontend and Backend allows the Compiler to perform some optimizations on a higher level. Examples are shown in the following sections.

Switch Statements

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. Table 8.6 describes how the Compiler implements these strategies.

Table 8.6 Switch Implementations

Method	Description
Branch Sequence	For small switches with scattered case label values, the Compiler generates an if ... elsif ... elsif ... else ... sequence if the Compiler switch -Os is active.
Branch Tree	For small switches with scattered case label values, the Compiler generates a branch tree. This is 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.

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.

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.

Using Qualifiers for Pointers

The use of qualifiers (`const`, `volatile`, ...) for pointers is confusing. 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;
}
```

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:

```
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.

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 8.42).

Listing 8.42 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 8.43 Invoking the ClearR0 macro.

```
  ClearR0;
```

And then the preprocessor expands the macro.

Listing 8.44 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 8.45).

Listing 8.45 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 (Listing 8.46).

Listing 8.46

```
  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 8.47 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 8.48).

Listing 8.48 Calling the ClearR0andR1 macro

```
ClearR0andR1;
```

The preprocessor expands the macro (Listing 8.49).

Listing 8.49 Preprocessor expansion of the ClearR0andR1 macro.

```
{__asm CLR R0; __asm CLR R1; };
```

Using Macro Parameters

An HLI assembler macro may have some parameters which are referenced in the macro code. Listing 8.50 defines the `Clear1` macro that uses the var parameter.

Listing 8.50 Clear1 macro definition.

```
/* This macro initializes the specified variable to 0.*/
#define Clear1(var) {__asm CLR var;}
```

Listing 8.51 Invoking the Clear1 macro in the source code

```
Clear1(var1);
```

Listing 8.52 The preprocessor expands the Clear1 macro

```
{__asm CLR var1 ; };
```

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, the Compiler is instructed that in all the macros defined afterward, the instructions should remain unchanged wherever the symbol # is specified. This macro is valid for the rest of the file in which it is specified.

Listing 8.53 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 8.54 Invoking the Clear2 macro in the source code

```
Clear2(var1);
```

Listing 8.55 The preprocessor expands the Clear2 macro

```
{ __asm LOAD #0,Reg0;__asm STORE Reg0,var1; };
```

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 Listing 8.56.

Listing 8.56 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 8.57 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.

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 (;....).

General

A header file must be specially prepared to generate the assembler include file.

Listing 8.58 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. Macros containing elements not supported by the assembler should be 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`.

Example

Listing 8.59 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 8.60).

Listing 8.60 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 8.61 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.

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 (Listing 8.62) shows some uses of this feature:

Listing 8.62 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 Listing 8.62 produces the following output (Listing 8.63):

Listing 8.63 Assembler listing of Listing 8.62

```
                        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 (Listing 8.64) do not work or are not generated:

Listing 8.64 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 Listing 8.64 produces the following output (Listing 8.65):

Listing 8.65 Assembler listing of Listing 8.64

```
                        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 just defined with no actual value as Nothing 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.

enums

enums in C have a unique name and a defined value. They are simply generated by the compiler as an `EQU` directive.

Listing 8.66 enum

```
#pragma CREATE_ASM_LISTING ON
enum {
  E1=4,
  E2=47,
  E3=-1*7
};
```

Creates:

Listing 8.67 Resultant EQUs from enums

```
E1                       EQU $4
E2                       EQU $2F
E3                       EQU $FFFFFFF9
```

NOTE *Negative values are generated as 32-bit hex numbers.*

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 typedef's 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 8.68 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 8.69 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.

Functions

Declared functions are generated by the `XREF` directive. This enables them to be used with the assembler. The function to be called from C, but defined in assembler, should not be generated 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 Listing 8.70.

Listing 8.70 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 8.71 Functions defined in assembler

```
    XREF  main
    XREF  f_C
```

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 8.72 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 8.73 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.

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 Listing 8.74 for an example of how C source code with its comments is converted into HC12 assembly.

Listing 8.74 C source code conversion to HC12 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
```

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 which should be translated to the assembler include file.

The created assembler include file then holds the corresponding assembler directives.

- The `-La` option should not be 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.

9

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.

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.

-Or: Register Optimization

When accessing pointer fields, this option prevents the compiler from reloading the address of the pointer for each access. An index register holds the pointer value over statements where possible.

NOTE This option may not be available for all targets.

-Oi: Inlining: Inline Functions

Use the inline keyword or the command line option -Oi for C/C++ functions. Defining a function before it is used helps the Compiler to inline it:

```
/* OK */
void foo(void);
void main(void) {
   foo();
}
void foo(void) {
  // ...
}
```

```
/* better! */
void foo(void) {
  // ...
}
void main(void) {
    foo();
}
```

This also helps the compiler to use a relative branch instruction instead an absolute.

__SHORT_SEG Segments

Variables allocated on the direct page (between 0 and 0xFF) are accessed using the direct addressing mode. The Compiler will allocate some variables on the direct page if they are defined in a `__SHORT_SEG` segment (Listing 9.1).

Listing 9.1 Allocate frequently-used variables on the direct page

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
unsigned int myvar3, myVar4.
```

In the previous example, ‘myVar1’ and ‘myVar2’ are both accessed using the direct addressing mode. Variables ‘myVar3’ and ‘myVar4’ are accessed using the extended addressing mode.

When some exported variables are defined in a `__SHORT_SEG` segment, the external declaration for these variables must also specify that they are allocated in a `__SHORT_SEG` segment. The External definition of the variable defined above looks like:

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4
```

The segment must be placed on the direct page in the PRM file (Listing 9.2).

Listing 9.2 Linker parameter file

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM                   INTO  MY_ROM;
    DEFAULT_RAM                   INTO  MY_RAM;
    _ZEROPAGE, myShortSegment      INTO  Z_RAM;
END
STACKSIZE 0x60
```

```
VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE The linker is case-sensitive. The segment name must be identical in the C and PRM files.

Defining I/O Registers

The I/O Registers are usually based at address 0. In order to tell the compiler it must use direct addressing mode to access the I/O registers, these registers are defined in a __SHORT_SEG section (if available) based at the specified address.

The I/O register is defined in the C source file as in Listing 9.3.

Listing 9.3 Definition of an I/O Register

```
typedef struct {
   unsigned char SCC1;
   unsigned char SCC2;
   unsigned char SCC3;
   unsigned char SCS1;
   unsigned char SCS2;
   unsigned char SCD;
   unsigned char SCBR;
} SCIStruct;
#pragma DATA_SEG __SHORT_SEG SCIRegs
SCIStruct   SCI;
#pragma DATA_SEG DEFAULT
```

Then the segment must be placed at the appropriate address in the PRM file (Listing 9.4).

Listing 9.4 Linker parameter file Allocating the I/O Register

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    SCI_RG = READ_WRITE  0x0013 TO 0x0019;
    Z_RAM  = READ_WRITE  0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE  0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY   0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM                    INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
    _ZEROPAGE                          INTO  Z_RAM;
    SCIRegs                            INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE The linker is case-sensitive. The segment name must be identical in the C/C++ and PRM files.

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.

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 */
}
```

HLI Assembly

Do not mix High-level Inline (HLI) Assembly with C declarations and statements (see Listing 9.5). 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, of course).

Listing 9.5 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 9.6).

Listing 9.6 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 */
}
```

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:

Example

```
i = 3;

foo(i++, --i);
```

In the above example, foo() is called either with ‘foo(3,3)’ or with ‘foo(2,2)’.

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.

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).

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.

Struct Returns

Normally the compiler has first to allocate space on the stack for the return value (1) and then to call the function (2). Phase (3) is for copying the return value to the variable s. In the callee ‘foo’ during the return sequence, the Compiler has to copy the return value (4, struct copy).

Depending on the size of the struct, this may be done inline. After return, the caller ‘main’ has to 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 use a lot of execution time, and this could mean a lot of code and stack usage.

Listing 9.7 Returning a struct can force the Compiler to produce lengthy code.

```
struct S foo(void)

  /* ... */
  return s; // (4)
}
```

```
void main(void) {
  struct S s;
  /* ... */
  s = foo();  // (1), (2), (3)
  /* ... */
}
```

With the example in Listing 9.8, the Compiler just has to pass the destination address and to call ‘foo’ (2). On the callee side, the callee copies the result indirectly into the destination (4). This approach reduces stack usage, avoids copying structs, and results in denser code. Note that the Compiler may also inline the above sequence (if supported). But for rare cases the above sequence may not be exactly the same as returning the struct by value (e.g., if the destination struct is modified in the callee).

Listing 9.8 A better way is to pass only a pointer to the callee for the return value.

```
void foo(struct S *sp) {
  /* ... */
  *sp = s; // (4)
}
void main(void) {
  S s;
  /* ... */
  foo(&s);  // (2)
  /* ... */
}
```

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 stack usage.

Stack access capabilities of the target influences the code quality. Depending on the target capabilities, access to local variables may be very inefficient. A reason might be the lack of a dedicated stack pointer (another address register has to be used instead, thus it might not be used for other values) or access to local variables is inefficient due the target architecture (limited offsets, only few addressing modes).

Allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the stack frame in the beginning of the function and to deallocate them in the exit part (Listing 9.9):

Listing 9.9 Good candidate for global variables

```
void foo(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 Compiler may offer an option to overlap parameter or local variables using a technique called ‘overlapping’. Local variables or parameters are allocated as global ones. The linker overlaps them depending on their use. For targets with limited stack (e.g., no stack addressing capabilities), this often is the only solution. However this solution makes the code non-reentrant (no recursion is allowed).

Parameter Passing

Avoid parameters which exceed the data passed through registers (see Backend).

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.

Inlining and Macros

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++);

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 9.10 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 Listing 9.11 is much simpler and faster:

Listing 9.11 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.

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.

Short Segments

Whenever possible and available (not all targets support it), place frequently used global variables into a `DIRECT` or `__SHORT_SEG` segment using

```
#pragma DATA_SEG __SHORT_SEG MySeg
```

Qualifiers

Use the ‘`const`’ qualifier to help the compiler. The ‘const’ objects are placed into ROM for the HIWARE object-file format if the -Cc compiler option is given.

10

HC(S)12 Backend

The Backend is the target–dependent part of a Compiler containing the code generator. This chapter discusses the technical details of the Backend for the M68HC(S)12 family.

The HC(S)12 backend chapter covers these sections:

Memory Models

This section describes the following memory models:

SMALL memory model

The Compiler for the MC68HC(S)12 supports three different memory models. The default is the SMALL memory model, which corresponds to the normal setup, i.e., a 64 kB code-address space. If you use a code-memory expansion scheme, you may use the BANKED memory model. The LARGE memory model supports both data and code expansion. The different memory models change the default behavior of the compiler.

BANKED memory model

Some microcontrollers of the M68HC12 family have the ability to extend the address range of the CPU beyond the 64kB limit given by the 16 CPU address lines. This feature is provided by a paging scheme using expansion address lines. The exact method to extend the address space is hardware-dependent.

There are several expansion memory banks. Which bank is active is determined by the value of a dedicated I/O register in memory (page register). Part of the memory is non–banked, accessible from all expansion memory banks.

The *BANKED* memory model is identical to the *SMALL* memory model in terms of variable allocation. Part of your code may be allocated to extended memory, thus breaking the 64 kB limit.

If a function is in extended memory, it has to be called differently than a function in non–banked memory. In particular, a bank switch has to be done:

- The current bank number has to be saved
- The called function's bank number has to be written to the bank register (bank switch)
- The function has to be called.

__far and __near for functions

In order to minimize overhead, functions are separated into two classes: `__far` functions are always called with a `CALL`, while `__near` functions are simply called with a `JSR/BSR`. If a `__near` function is called, the callee must be either in non–banked memory, or in the same memory bank as the caller.

When compiling in the *BANKED* or the *LARGE* memory model, all default functions are `__far`. To override this default, explicitly declare a function as `__near` or `__far`, for example:

```
static int __far my_func (int *p);
```

In the *BANKED* or in the *LARGE* memory model, function pointers are always 24 bits wide. The page is allocated differently for 24-bit function pointers than for 24-bit `__far` data pointers. For a 24-bit function pointer, the page is allocated at an offset of 2 bytes. This difference is because of hardware requirements.

Table 10.1 shows the allocation for a banked function pointer:

Table 10.1 Banked function-pointer allocation

Byte 0	Byte 1	Byte 2
offset highbyte	offset lowbyte	page

Table 10.2 shows the allocation for a `__far` data pointer.:

Table 10.2 __far data pointer allocation

Byte 0	Byte 1	Byte 2
page	offset highbyte	offset lowbyte

The compiler does not exchange the byte order when assigning a `__far` function pointer to `__far` data pointer or a `__far` data pointer to `__far` function pointer. The special byte ordering is also not automatically adapted when using absolute addresses for `__far` function pointers.

The following two macros (Listing 10.1 and Listing 10.2) can be used to manually assign and adapt one `__far` data pointer to a `__far` function pointer (or vice versa). See Listing 10.3.

Listing 10.1 CONV_FAR_FUN_TO_DATA_PTR macro

```
#define CONV_FAR_FUN_TO_DATA_PTR(to, from)\
  *(int*)((char*)&to+1) = *(int*)&from; \
  *(char*)&to = *((char*)&from+2);
```

Listing 10.2 CONV_FAR_DATA_TO_FUN_PTR macro

```
#define CONV_FAR_DATA_TO_FUN_PTR(to, from)\
  *(int*)&to = *(int*)((char*)&from+1);\
  *((char*)&to+2) = *(char*)&from;
```

Listing 10.3 Using the CONV_FAR_FUN_TO_DATA_PTR and CONV_FAR_DATA_TO_FUN_PTR macros

```
#pragma CODE_SEG __PIC_SEG __NEAR_SEG PIC_CODE
void __far Function(void) {
}
void __far NextFun(void) {}
#pragma CODE_SEG DEFAULT
char RamBuf[100];
void Test(void) {
  void (*__far startFunPtr)(void)= Function;
  void (*__far endFunPtr)(void)= NextFun;
  void (*__far bufferFunPtr)(void);
  char *__far startDataPtr;
  char *__far endDataPtr;
  char *__far bufferDataPtr= RamBuf;
  int i=0;
  CONV_FAR_FUN_TO_DATA_PTR(startDataPtr, startFunPtr);
  CONV_FAR_FUN_TO_DATA_PTR(endDataPtr, endFunPtr);
  CONV_FAR_DATA_TO_FUN_PTR(bufferFunPtr, bufferDataPtr);
  while (startDataPtr != endDataPtr) {
    RamBuf[i++]= *(startDataPtr++);
  }
  bufferFunPtr();
}
```

NOTE In the previous example, code is executed at a different place than it was linked. Therefore, this code must be compiled position-independent. However, PIC code is not supported for the bank part of the address.

NOTE The different byte ordering only causes problems with the __far function pointer. With the __near calling convention, straightforward code can be used. Also as PIC is only supported inside of one bank, PIC code is usually using the __near calling convention.

See also the Position-Independent Code (PIC) section and the -Pic: Generate Position-Independent Code (PIC) compiler option

Non–Banked Memory

Some parts of an application must always be in non–banked memory, in particular:

- The prestart code (`_PRESTART` segment)
- The startup code (`NON_BANKED` segment) and the startup descriptors (`STARTUP` segment)
- All runtime support routines (`NON_BANKED` segment)
- All interrupt handlers, because trap vectors are only 16 bits wide.

For more information on these segments, see the Linker section in the Build Tools manual.

Usually, some initial settings are necessary to enable the memory expansion scheme. You might want to include this initialization code in the startup function.

Using the Banked Memory Model

When the banking memory model is used, some constraints apply to the application's linker parameter files.

Definition of the Application Memory Map

The `SECTIONS` block in the PRM file contains the memory area definitions that are used by the application. A typical `SECTIONS` block for a banked application contains at least one definition for following memory blocks:

- One or several sections for the RAM area
- One section for the non-banked ROM area
- One section for each bank used by the application.

Banking is performed through a window. The size or start addresses depend on the hardware. The address space for each bank is defined the following way in the linker PRM file:

```
0x<bnr><startAddr> TO <bnr><endAddr>
```

where:

- `bnr` is the bank number.

  The value of this number depends on the hardware and is the bit pattern to be written into the bank register to access this bank. Valid values depend on the hardware configuration.
- `startAddr`: is the start address of the bank window. This has to be 4 hex digits.
- `endAddr`: is the end address of the bank window (inclusive; 4 hex digits).

In the following example, it is assumed that the bank window is defined between address `0x8000` and `0xBFFF` (Listing 10.4):

Listing 10.4 Example of a SECTIONS block in a PRM file

```
SECTIONS
    DIRECT_RAM      = READ_WRITE   0x00000 TO 0x000FF;
    RAM_AREA        = READ_WRITE   0x00800 TO 0x00BFF;
    BANK_0          = READ_ONLY    0x08000 TO 0x0BFFF;
    BANK_1          = READ_ONLY    0x18000 TO 0x1BFFF;
    BANK_2          = READ_ONLY    0x28000 TO 0x2BFFF;
    BANK_3          = READ_ONLY    0x38000 TO 0x3BFFF;
    NON_BANKED_ROM = READ_ONLY    0x0C000 TO 0x0FFFF;
```

Segment Allocation

Some predefined sections must be allocated in the NON_BANKED memory area otherwise the application will not be able to run correctly. The following predefined sections must always be located in the non-banked ROM memory area:

- `_PRESTART`: Contains the application's prestart code.
- `STARTUP`: Contains the application's startup structure
- `ROM_VAR`: Contains the application's constant variables
- `STRINGS`: Contains the application's string constants
- `COPY`: Contains the initialization values for the application's variables.
- `NON_BANKED`: Contains the run-time library functions.

In addition, as banked memory is only available for code sections (sections containing functions), all user-defined data or constant segments must be located on the non-banked memory area.

As the entry in the vector table is only two bytes wide, all the interrupt functions must also be allocated in the non-banked memory area.

In the following example (Listing 10.5), it is assumed that the bank window is defined between address `0x8000` and `0xBFFF`.

Listing 10.5 Example PRM file

```
LINK test.abs

NAMES test.o ansib.lib start12b.o END
SECTIONS
    DIRECT_RAM      = READ_WRITE   0x00000 TO 0x000FF;
    RAM_AREA        = READ_WRITE   0x00800 TO 0x00BFF;
    BANK_0          = READ_ONLY    0x08000 TO 0x0BFFF;
    BANK_1          = READ_ONLY    0x18000 TO 0x1BFFF;
    BANK_2          = READ_ONLY    0x28000 TO 0x2BFFF;
    BANK_3          = READ_ONLY    0x38000 TO 0x3BFFF;
    NON_BANKED_ROM = READ_ONLY    0x0C000 TO 0x0FFFF;
PLACEMENT
    _PRESTART, STARTUP, ROM_VAR,
    STRINGS, NON_BANKED,
    Int_Function,
    COPY                       INTO NON_BANKED_ROM;
    DEFAULT_RAM                INTO RAM_AREA;
    UserSeg1, UserSeg2,
    UserSeg3, DEFAULT_ROM      INTO BANK_0, BANK_1, BANK_2, BANK_3;
END
STACKSIZE 0x50
```

According to the previous PRM file:

- The NON_BANKED_ROM section contains the six predefined sections enumerated in the PLACEMENT block plus the Int_Function segment. The user-defined code segment, 'Int_Function', is where all the interrupt functions are allocated.
- The RAM_AREA section contains all the linker predefined and user-defined data segments, as well as the stack.
- The BANK_0, BANK_1, BANK_2, and BANK_3 sections contain the DEFAULT_ROM segment, as well as the user-defined code 'UserSeg1', 'UserSeg2', and 'UserSeg3' segments.
- The linker allocates first all functions implemented in the 'UserSeg1' segment, then the functions from 'UserSeg2', then the functions from 'UserSeg3', and finally the functions defined in the other segments.
- For the allocation of the functions, the linker first uses the BANK_0 section. As soon as this section is full, allocation continues in the BANK_1 section, then in BANK_2, and so on until all the functions are allocated. During the allocation, a specific function is always allocated on a single bank.

Simple example for the HC12DG128

A simple example for the HC12DG128 is shown below. The application uses three code banks:

- Bank 1 contains the code (read-only)
- Bank 2 contains constant initialized data (read-only, MyConstSegPage2)
- Bank 3 contains constant initialized data (read-only, MyConstSegPage3)

The source for this is (Listing 10.6):

Listing 10.6 Banked-memory example for the HC12DG128

```
/* bankcnst.c */
#pragma CONST_SEG __PPAGE_SEG MyConstSegPage2
volatile const int aa = 3;
#pragma CONST_SEG __PPAGE_SEG MyConstSegPage3
```

```
volatile const int xx = 2;
#pragma CONST_SEG DEFAULT
void main(void) {
  volatile int cc = xx+aa;
}
```

All variables are declared as `volatile` to avoid the compiler optimizing many accesses. The above source is compiled with following Compiler command line:

```
bankcnst.c -F2 -CpPpage=RUNTIME -Mb
```

- The ELF/DWARF Object File Format is chosen with `-F2`
- `-CpPage=RUNTIME` is used because we are accessing other `PPAGE` constant data (`MyConstSegPage2`, `MyConstSegPage3`) from the code page (page 1). We use a runtime routine to switch the (code) pages. This runtime routine has to be placed in a non-banked area.
- `-Mb` tells the compiler to use the banked-memory model.

The startup module and the data page module must be recompiled because they are not delivered by default with the above-listed configuration/option settings:

```
datapage.c start12.c -F2 -CpPpage=RUNTIME -Mb -DDG128
```

- The reasons for `-F2`, `-CpPpage`, and `-Mb` are listed above.
- The option `-DDG128` is not for the startup code, it is for `datapage.c`. Because of this define, `datapage.c` is aware that the page register is at `0xff`. And `datapage.c` also uses a more efficient version which only considers one page register.
- Recompiling `datapage.c` is necessary because the page register is for the DG128 at a different location.
- The `start12.c` startup code is recompiled even if it is not really necessary here. Recompiling `start12.c` is necessary only because the startup code does not initialize variables in pages by default in the small or banked memory models. In the example above, the initialized variables are constant and thus initialized during downloading.
- Note that some segments in the prm file must not be in a paged area (e.g., `NON_BANKED`).

Now the application is linked. In the linker parameter file in Listing 10.7, all three pages are declared. The Bank Window for `PPAGE` is in the range of `0x8000` to `0xBFFF`:

Listing 10.7 PRM file for previous example

```
LINK bankcnst.abs

NAMES bankcnst.o datapage.o start12.o ansib.lib END

SECTIONS
    MY_RAM   = READ_WRITE 0x800   TO 0x80F;
    MY_ROM   = READ_ONLY  0x810   TO 0xAFF;
    MY_PAGE1 = READ_ONLY  0x18000 TO 0x1BFFF;
    MY_PAGE2 = READ_ONLY  0x28000 TO 0x2BFFF;
    MY_PAGE3 = READ_ONLY  0x38000 TO 0x3BFFF;
    MY_STK   = READ_WRITE 0xB00   TO 0xBFF;
END

PLACEMENT
    DEFAULT_ROM       INTO  MY_PAGE1;
    MyConstSegPage2   INTO  MY_PAGE2;
    MyConstSegPage3   INTO  MY_PAGE3;
    _PRESTART, STARTUP,
    ROM_VAR, STRINGS,
    NON_BANKED, COPY  INTO  MY_ROM;
    DEFAULT_RAM       INTO  MY_RAM;
    SSTACK            INTO  MY_STK;
END

VECTOR 0 _Startup
```

Finally, load the application into the simulator to simulate it, or download it onto the HC12DG128.

LARGE memory model

The default large memory model supports both extended data and code. See the BANKED memory model section for code-banking constraints. See the Paged Variables section for data-paging support.

Because paged variables are not directly supported by the HC(S)12 instruction set, the LARGE memory model has significant overhead compared with the SMALL or BANKED memory models.

Note that `__far` functions and paged variables are possible in all memory models. If they are not defaulted to by the memory model, the code is adapted to use these features. If only a small part of the application actually needs paged variables, for example, then using a smaller memory model and adapting the small model generates smaller and faster code.

Implicit __near pointer conversions

In the large memory model, the stack pointer is 16 bits wide. The default allocation for any objects on the stack is `__near`. In the example in Listing 10.8, `i_global` is accessed with a `__far` access, while `i_local` is accessed directly.

Listing 10.8 Example with both __near and __far memory accesses

```
int i_global;
void main(void) {
  int i_local;
}
```

The HC12 casts `__near` pointers to standard pointers for all implicit parameter declarations and for open parameter arguments. The following code in Listing 10.9 will only work with this extension:

Listing 10.9 Example with implicit parameter declaration

```
void main(void) {
  int i;
  sscanf("3","%d",&i);
}
```

NOTE The size of a __near pointer only differs in the LARGE memory model from the size of the standard pointer type. Therefore, applications using the SMALL or BANKED memory models are not similarly affected.

Non-ANSI Keywords

Table 10.3 gives an overview about the supported non-ANSI keywords:

Table 10.3 Supported non-ANSI Keywords

Keyword	Data Pointer	Supported for Function Pointer	Function
__far	yes	supported for ELF or BANKED/LARGE Memory model	yes
__near	yes	supported for ELF or SMALL memory model	yes
__dptr (valid with -cpuhcs12x option)	yes	yes	no
__rptr (valid with -cpuhcs12x option)	yes	yes	no

Table 10.3 Supported non-ANSI Keywords (*continued*)

Keyword	Data Pointer	Supported for Function Pointer	Function
__eptr (valid with -cpuhcs12x option)	yes	yes	no
__pptr (valid with -cpuhcs12x option)	yes	yes	no
interrupt	no	no	yes

Data Types

This section describes how the basic types of ANSI-C are implemented by the MC68HC(S)12 Backend.

Scalar Types

All basic types may be changed with the -T: Flexible Type Management compiler option. All scalar types (except char) are without a signed/unsigned qualifier, and their default values are signed (e.g., ‘int’ is the same as ‘signed int’).

Table 10.4 gives the sizes of the simple types together with the possible formats using the -T option.

Table 10.4 Types and Formats for the -T Option

Type	Default Format	Default Value Range		Formats available with the -T Option
		Min	Max	
char (unsigned)	8-bit	0	255	8-, 16-, & 32-bit
signed char	8-bit	-128	127	8-, 16-, & 32-bit
unsigned char	8-bit	0	255	8-, 16-, & 32-bit
signed short	16-bit	-32,768	32,767	8-, 16-, & 32-bit
unsigned short	16-bit	0	65,535	8-, 16-, & 32-bit
enum (signed)	16-bit	-32,768	32,767	8-, 16-, & 32-bit
signed int	16-bit	-32,768	32,767	8-, 16-, & 32-bit

Table 10.4 Types and Formats for the -T Option (*continued*)

Type	Default Format	Default Value Range		Formats available with the -T Option
		Min	Max	
unsigned int	16-bit	0	65,535	8-, 16-, & 32-bit
signed long	32-bit	-2,147,483,648	2,147,483,647	8-, 16-, & 32-bit
unsigned long	32-bit	0	4,294,967,295	8-, 16-, & 32-bit
signed long long	32-bit	-2,147,483,648	2,147,483,647	8-, 16-, & 32-bit
unsigned long long	32-bit	0	4,294,967,295	8-, 16-, & 32-bit

NOTE Plain type `char` is unsigned. This default can be changed by the `-T` option.

Floating-Point Types

The Compiler supports the two IEEE standard formats (32 and 64 bits wide) for floating point types. By default, the Compiler uses the IEEE32 format both for float and double.

The -T: Flexible Type Management option may be used to change the default format of float/double.

Table 10.5 Floating-Point Representation

Type	Default Format	Default Value Range		Formats Available With the -T Option
		Min	Max	
float	IEEE32	-1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64

Table 10.5 Floating-Point Representation (*continued*)

Type	Default Format	Default Value Range		Formats Available With the -T Option
		Min	Max	
long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
long long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64

Pointer Types and Function Pointers

The size of pointer types depends on the memory model selected. Table 10.6 gives an overview.

Table 10.6 Pointer sizes

Type	Example	Size		
		SMALL	BANKED	LARGE
default data pointer	char*	2 bytes	2 bytes	3 bytes
__near data pointer	char* __near	2 bytes	2 bytes	2 bytes
__far data pointer	char* __far	3 bytes	3 bytes	3 bytes
default function pointer	void (*)(void)	2 bytes	3 bytes	3 bytes
__near function pointer	void (*__near)(void)	2 bytes	2 bytes*(1)*	2 bytes*(1)*
__far function pointer	void (*__far)(void)	3 bytes*(1)*	3 bytes	3 bytes

(1): Only supported in the ELF Object File Format.

Pointer Arithmetic

The HCS12(X) compiler performs 16-bit pointer arithmetic. The page part of a 24-bit pointer type is not affected by pointer arithmetic, therefore it is not possible to change the referenced page by incrementing or decrementing a 24-bit pointer. When using pointer arithmetic, keep these items in mind:

- Do not allocate objects in more than one page (i.e., the must not cross a page boundary). The linker does not split objects into multiple pages.
- Write special assembly routines or macros to use 24-bit pointer arithmetic.
- Split larger objects into multiple parts.
- The maximum object size is:
  - 64 kilobyte for `GPAGE`
  - 16 kilobyte for `PPAGE`
  - 4 kilobyte for `RPAGE`
  - 1 kilobyte for `EPAGE`

Listing 10.10 Pointer Arithmetic Example

```
char array[1000];
int i;
char *__far pf = array;
...
for (i = 0; i < 1000; i++) *(pf++) = 0
...
```

In this preceding example, the global variable `array` must be located in one single page.

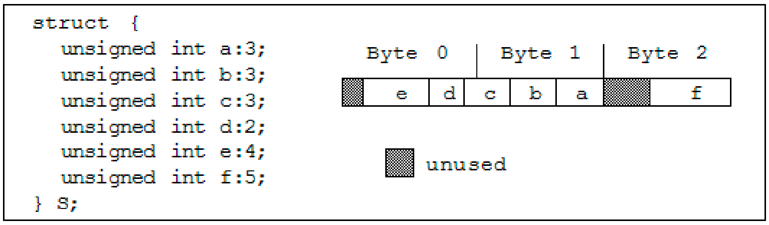
Structured Types, Alignment

Local variables are allocated on the stack (which grows downwards). The order of allocation of local variables depends on how often the variables are used. More often used variables are closer to the stack top. This reordering is done to take advantage of the shorter index addressing modes. The most significant part of a simple variable always is stored at the low memory address (big endian).

Bitfields

The maximum width of bitfields is 32 bits. The allocation unit is a byte. The Compiler uses words only if a bitfield is wider than eight bits, or if using bytes would cause more than two unused bits. Allocation order is from the least significant bit up to the most significant bit in the order of declaration. Figure 10.1 illustrates this allocation scheme.

Figure 10.1 Bitfield allocation scheme



```
struct {
  unsigned int a:3;
  unsigned int b:3;
  unsigned int c:3;
  unsigned int d:2;
  unsigned int e:4;
  unsigned int f:5;
} S;
```

Paged Variables

The HC(S)12 has several page registers that control different areas of the 64 kB address space. The following table gives an overview about the page register names, their memory addressing capabilities and their default location.

Table 10.7 Page-register memory areas for HC12 A4

Page Register	Start Address	End Address	Default Port Address
DPAGE	0x7000	0x7FFF	0x34
EPAGE	0x0400 or 0x0000	0x07FF or 0x03FF	0x36
PPAGE	0x8000	0xBFFF	0x35

Table 10.8 Page-register memory areas for HCS12X DP series

Page Register	Start Address	End Address	Default Port Address
PPAGE	0x8000	0xBFFF	0x30
RPAGE	0x1000	0x1FFF	0x16
EPAGE	0x0800	0x0BFF	0x17

The Compiler supports variable accesses with the DPAGE, the EPAGE, and with the PPAGE page registers. Variables in paged memory areas must be defined after one of the following pragmas:

```
#pragma DATA_SEG __DPAGE_SEG segment_name

#pragma DATA_SEG __EPAGE_SEG segment_name

#pragma DATA_SEG __PPAGE_SEG segment_name

#pragma DATA_SEG __RPAGE_SEG segment_name

#pragma DATA_SEG __GPAGE_SEG segment_name
```

Segment_name must be allocated with the Linker at a memory area which is controlled by the corresponding page register.

The Compiler supports the `__far` data pointer, which may point to all variables independent of their page register. The `__far` keyword is written immediately after the "*".

Example:

```
#pragma DATA_SEG __DPAGE_SEG my_DPAGE

int a; /* variable in memory controlled by DPAGE reg.*/

int*__far p = &a; /* __far pointer to access any variable*/
```

For the following topics, the Compiler must know which page register is used for data paging:

- interrupt routines:

  An interrupt routine saves, by default, those page registers given by the command line option "`-Cp`".

- `__far` data pointer accesses

  If only one page register is used, then the __far data pointer accesses is inlined because the page register is obvious. If several page registers are possible, a runtime routine determines the correct page register. The page register is determined from the offset portion of the address.

The are two ways to access a variable in paged memory:

- Store the page into the page register and then perform the usual assembler instructions.
- Use a runtime routine.

The first method is faster and denser than the second. If code and paged variables are in memory areas that are controlled by the same page register, the page register must not be modified. In this case a runtime routine for memory accesses must be used. Runtime routines must be in a non-paged memory area. By default, the first method is used except for PPAGE accesses in the *BANKED* memory model where a runtime routine is used.

```
-CpDPAGE["="(address|"RUNTIME")]

-CpEPAGE["="(address|"RUNTIME")]

-CpPPAGE["="(address|"RUNTIME")]
```

Example:

`-CpDPAGE=0x34`

Variable accesses to DPAGE segments are inlined. The address `0x34` is also a built-in default, so “`-CpDPAGE`” is equivalent to this argument.

NOTE With the form `-CpDPAGE=0x34` is implied that code could be inlined. If a runtime routine must be taken, then the address is not necessary.

To use a different page address than the default, the `datapage.c` library file must be adapted. It contains a define for the specific page register address. To use a modified `datapage.c` file, compile it with the correct options set and then specify the generated object file in front of the ANSI library in the link parameter file’s `NAMES` section.

Example:

`-CpEPAGE=RUNTIME`

Variable accesses to the EPAGE segments are done with a runtime routine.

NOTE The runtime routine is adapted to special requirements. The runtime routines are written for the most general case. If only one PAGE register is used, the runtime routines could be made faster and shorter. The runtime routines for paged data memory access are in the `datapage.c` file. Take care to implement the same interface, i.e., to save all registers as stated in the source code. Especially when using a RUNTIME access, which is the default in the large memory model. Adapting the `datapage.c` file’s routines can result in a time improvement of a factor of 2 or more.

Example:

```
    -CpPPAGE
```

In the *SMALL* memory model, variable accesses to PPAGE segments are inlined. No code of this compilation unit must be linked between `0x8000` and 0xbfff. In the *BANKED* memory model variable accesses are done with a runtime routine. Therefore there are no restrictions in linking the code between `0x8000` and `0xbfff`.

NOTE The Compiler defines the macros `__RPAGE__`, `__GPAGE__`, `__DPAGE__`, `__EPAGE__` and `__PPAGE__` if the corresponding compiler options are used.

For example, consider the following situation:

- The page registers are mapped to `0x2000` to be able to use the zero page.
- Variables are placed from 0x7000 up to 0x9FFF in different pages using the DPAGE and the PPAGE register.

- The code is placed from 0x2000 up to 0x7000 and from 0x9fff up to 0bfff.

The area controlled by the PPAGE register is used for functions and for variables. The following Compiler options should be used:

```
-CpDPAGE=0x2034 -CpPPAGE=RUNTIME
```

Variable accesses to the DPAGE are also done with the runtime routine, but the code is larger. Variable accesses to the PPAGE must be done with a runtime routine.

NOTE Several page registers could be used for data paging in the same compilation unit.

NOTE The `RUNTIME` option must be given in the SMALL memory model. In the `BANKED` and in the Large memory model it is the default and could be avoided. Nevertheless it is good practice to specify it.

Another point to consider about banked variables is the initialization. For the large memory model, paged variables are initialized correctly by default. In the small and banked memory models, the startup code and the Linker must be explicitly set up to use 24-bit addresses instead of 16-bit addresses. To produce startup code which handles 24-bit addresses, the startup code must be compiled with one of the Compiler options "`-Cp...`" as explained above.

For the HIWARE object file format, the Linker must be told to produce 24-bit addresses with the "`HAS_BANKED_DATA`" command in the link parameter file. For the ELF object file format, the Linker reads the size of the pointers for the startup structure by analyzing the debug info of the startup code. Only the startup code must be recompiled with the correct Compiler options for the ELF object file format.

The reason that the initialization for banked variables must be specified explicitly is that no overhead of banked data should occur as long as banked variables are not used.

By default the compiler assumes that objects in the default segment are distributed into different pages. However, objects in user-defined segments are on only one page. This behavior is changed with the -PSeg: Assume Objects are on Same Page compiler option.

NOTE The HCS12X architecture has an enhanced data paging mechanism. For data paging on HCS12X devices see TN238 and TN240.

Position-Independent Code (PIC)

The HC(S)12 compiler supports position-independent code. PIC functions are larger and slower than non-pic functions, therefore PIC code should only be generated when necessary.

To compile one function as PIC, use the #pragma CODE_SEG: Code Segment Definition environment variable with the `__PIC_SEG` modifier. To compile one compilation unit as PIC, use the -Pic: Generate Position-Independent Code (PIC) compiler option. The pragma has the advantage that it allows PIC and non-PIC functions and function calls in the same compilation unit. With this option, all functions and all calls (except runtime routine calls) are position-independent (Listing 10.11).

Listing 10.11 Compiling a Function or Compilation Unit as PIC

```
#pragma CODE_SEG __PIC_SEG PIC_CODE
void f_PIC(void); /* declare f_PIC to be in specific PIC segment */

#pragma CODE_SEG DEFAULT
void f_NonPic(void) {
  f_PIC(); /* NON pic call, calls PIC function at link address only */
}

#pragma CODE_SEG __PIC_SEG PIC_CODE
void g_PIC(void) {}
int i;
void f_PIC(void) {
  if (i) {      /* global variables are accessed absolute   */
    g_PIC();    /* calls g_PIC relative to current location */
    f_NonPic(); /* calls function at link time address      */
  }
}
```

Listing 10.12 shows the disassembled code produced by the previous Listing.

Listing 10.12 Machine Code Generated by the Source Code Listed Above

```
f_NonPic:
  0000 060000       JMP   f_PIC

g_PIC:
  0000 3d           RTS

f_PIC:
  0000 fc0000       LDD   i
  0003 2707         BEQ   *+9 ;abs = 000c
  0005 15fa0000     JSR   g_PIC,PCR
  0009 160000       JSR   f_NonPic
  000c 3d           RTS
```

Note that the call from the `f_PIC` PIC function to the `g_PIC` PIC function is done with a PC-relative `JSR` instead of an shorter extended `JSR`. The calls from the non-PIC function to `f_PIC` and back are encoded with absolute calls.

Taking the address of a function returns the link time address of this function. Listing 10.13 shows a small application that copies a part of itself into RAM. Then the RAM copy is started and executed until a `HALT` occurs (which is implemented with a HC12 `SWI` instruction).

Listing 10.13 Taking the address of a function

```
#include <hidef.h>  /* for HALT    */
#include <string.h> /* for memmove */

#pragma CODE_SEG __PIC_SEG __NEAR_SEG PIC_CODE
/* declarations of PIC functions */
void f0(void);
void f1(void);
void f2(void);
/* implementation of PIC functions */
void f0(void) {
  /* here we calculate the address of the RAM copy of f1 */
  /* by using inline assembly */
  void (*pf1) (void);
  __asm LEAX  f1,pcr;
  __asm STX   pf1;
  pf1();
}
void f1(void) { /* just call f2 */
  f2();
}
void f2(void) {
  HALT;  /* finished, call the user/debugger */
}
void end(void){} /* dummy function to calculate the end of */
                 /* the PIC_CODE segment                   */

/* implementation of main module. Copies and starts the PIC code */
#pragma CODE_SEG DEFAULT
char buf[100]; /* RAM area into which to copy the PIC functions  */
```

```
void main(void) {
  /* copy PIC functions */
  memmove(buf, (char*)f0,(char*)end-(char*)f0);
  /* start f0 */
  ((void(*)(void))buf) (); /* cast buf to fnct pointer and call it */
}
```

Listing 10.14 shows the disassembled code from the previous Listing.

Listing 10.14 Machine Code Generated by the Source Code Listed Above

```
f0:
  0000 3b             PSHD
  0001 1afa0000       LEAX  f1,PCR
  0005 6e80           STX   0,SP
  0007 15f30000       JSR   [0,SP]
  000b 3a             PULD
  000c 3d             RTS

f1:
  0000 05fa0000       JMP   f2,PCR

f2:
  0000 c7             CLRB
  0001 3f             SWI
  0002 3d             RTS

end:
  0000 3d             RTS

main:
  0000 cc0000         LDD   #buf
  0003 3b             PSHD
  0004 ce0000         LDX   #f0
  0007 34             PSHX
  0008 cc0000         LDD   #end
  000b 830000         SUBD  #f0
  000e 160000         JSR   memmove
  0011 1b84           LEAS  4,SP
  0013 060000         JMP   buf
```

With the -Pic: Generate Position-Independent Code (PIC) compiler option, runtime functions are still called absolutely. In order to generate PIC runtime calls, use the additional -PicRTS: Call Runtime Support Position Independent compiler option.

The delivered libraries are not built position-independent. In order to move them together with your code, rebuild your code with the `-Pic -PicRts` compiler option. There is a make file to build the library. Please check the maker section in the Build Tools manual for details.

PIC Impacts on generated code:

- Absolute calls are encoded PC-relative. Calls via function pointers are not affected.
- Long branches are done with the `LBRA` instruction instead of an extended `JMP`.
- The indexed 16-bit Constant Indirect (`[IDX2]`) addressing mode using the PC register is not used by the compiler to access via absolute pointers.
- Switches are encoded by binary search trees instead of tables (which contain absolute addresses).

Restrictions

The compiler does not support position-independent data. To use position-independent data, a local variable or parameter pointing to a moveable structure containing all global data must be used. If the whole application, including constants, should be position-independent, this restriction has the following implications:

- The startup code accesses the global data structure _startupData absolutely. In order to build a completely PIC application, do not use this startup code. Without the startup code, global variables won't be initialized.
- Strings as in 'PutString(“Hello World”);' are considered as global data and can therefore not be moved together with the code. Use a pointer pointing to the actual string instead. For example, “PutString(dataPtr->hello_world);” with dataPtr set the actual position before.
- The debug info is only generated for the link time version of the functions. Without any debugger extension, copied PIC functions will not have debug info.
- Only __near (16-bit address space) functions are fully supported for PIC code. For calls to __far (24-bit) functions, only the 16-bit offset of the address is position independent. The page is hard encoded into the call instructions. Therefore __far functions can be moved in the same page and to a non-paged area. They cannot be moved into a different page.
- Runtime routine calls have are always `__near`. They are absolute, unless the -PicRTS compiler option is specified also. Note that the message C3605 is issued whenever a runtime routine is called. By setting this message to an error, you can check if your code uses runtime routines.
- Some ANSI routines are using global data, like the error variable “errno”. The memory allocation functions do access the global memory and the `strtok()` ANSI function also has a global state. These functions require to have fix placed data.

See also

Compiler options:

- -Pic: Generate Position-Independent Code (PIC)

- -PicRTS: Call Runtime Support Position Independent
- #pragma CODE_SEG: Code Segment Definition

Register Usage

The Compiler uses all registers of the MC68HC12 except the TMP2 and the TMP3 registers. These registers are never accessed from C code.

Call Protocol and Calling Conventions

This section covers the following topics:

- "Argument Passing" on page 533
- "Return Values" on page 534
- "Returning Large Results" on page 534

Argument Passing

The Pascal calling convention is used for functions with a fixed number of parameters: The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

The C calling convention is used for functions with a variable number of parameters. In this case, the caller pushes the arguments from right to left. If the last parameter of a function with a fixed number of arguments has a simple type, it is not pushed but passed in a register.

This results in shorter code because pushing the last parameter is saved. Table 10.9 gives an overview of the registers used for argument passing.

Table 10.9 HC(S)12 registers employed in passing arguments

Size of Last Parameter	Type Example	Register
1 byte	char	B
2 bytes	int, array	D
3 bytes	__far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Parameters having a type not listed above are passed on the stack (i.e., all types having a size greater than four bytes).

Return Values

Function results are returned in registers, except if the function returns a result larger than one word (see below). Depending on the return type, different registers are used (Table 10.10).

Table 10.10 HC(S)12 registers employed in function returns

Size of Return Value	Type Example	Register
1 byte	char	B
2 bytes	int	D
3 bytes	__far data pointer	X(L), B(H)
4 bytes	long	D(L), X(H)

Returning Large Results

Functions returning a result larger than two words are called with an additional parameter. This parameter is the address where the result should get copied.

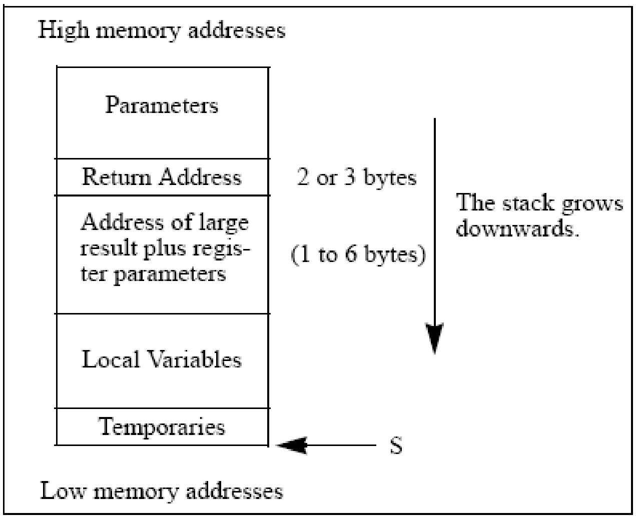
Stack Frames

Functions have a stack frame containing all their local data. The Compiler uses the stack pointer as the base address for accessing local data.

If one of the `NO_ENTRY`, `NO_EXIT`, or `NO_FRAME` pragmas is active, the Compiler does not generate code to set up a stack frame for this function. In this case the function must have neither local variables nor parameters.

Figure 10.2 shows the stack frame of a normal function, i.e., compiled with above pragmas inactive.

Figure 10.2 Normal stack frame



Entry Code

Normal *entry code* is a sequence of instructions reserving space for local variables and writing eventually the register parameter to the stack:

for a 1-byte register parameter:

```
PSHB
```

for a 2-byte register parameter:

```
PSHD
```

for a 3-byte register parameter:

```
PSHX
PSHB
```

for a 4-byte register parameter:

```
PSHD
PSHX
```

In addition, the entry code also allocates space for local variables. This may be done before or after the push for the register parameter. If it is done before the push of the register parameter, the push and the allocation may be optimized into a single store instruction with auto-decrement. Also, space for one or two bytes may be allocated by a push instruction instead of an `LEAS` to save space.

Exit Code

Exit code removes local variables from the stack before returning to the caller. The exit code is optimized depending on the `-Os` (optimize for size, default) or `-Ot` (optimize for time) compiler command-line switches:

```
                -Os                   -Ot

1 byte to release:
                PULA or PULB           LEAS  #1, SP

2 bytes to release:
                PULX, PULY or PULD     LEAS  #2, SP

3 bytes or more to release:
                LEAS  #size, SP       LEAS  #size, SP
```

If the `TRAP_PROC` pragma is active, then `RTC/RTS` is replaced by an `RTI` instruction.

Calling a __far Function

Calling a normal __far function is done with `CALL/RTC`. The return address for a __far function is three bytes large. The offset of parameters not passed in a register is one larger than for `__near` functions.

__far and __near

The `__near` and `__far` keywords enable you to control the calling convention (Listing 10.15).

Listing 10.15 __near and __far keywords

```
void __far f(void);
void __near g(void);
#pragma DATA_SEG __NEAR_SEG my_near_seg
```

```
void  h(void) {
  f();
  g();
}
```

The `h()` function is compiled with the `__near` calling convention, i.e., it ends with an `RTS` instruction. The call to `f()` is done with the `__far` calling convention, i.e., with a `CALL` instruction. The call to `g()` is done with the `__near` calling convention, i.e., with either a `BSR` or a `JSR` instruction. The difference between using the `__near` and the `__far` keywords to using the pragma is that the pragma also specifies a segment. With the `__far` keyword it is up to you to place a `__near` function at a reachable address.

The default calling convention depends on the memory model. It is `__near` for the SMALL memory model and `__far` for the BANKED memory model.

The `__far` keyword can also be used to specify a `__far` data pointer. The `__far` keyword is placed immediately after the “`*`” like the `const` type qualifier. If no `__far` keyword is used, a data pointer is 16 bits wide.

Pragmas

The Compiler provides a couple of pragmas that control the allocation of stack frames and the generation of entry and exit code.

TRAP_PROC

The procedure terminates with an `RTI` instruction instead of an `RTS`.

NO_ENTRY

Omits generation of procedure entry code.

NO_EXIT

Does not generate procedure exit code. It's the programmer's responsibility that the function returns somehow!

NO_FRAME

No stack frame is set up, but the Compiler generates an `RTS/RTC` (or `RTI`, if the `TRAP_PROC` pragma is active).

Interrupt Functions

For interrupt procedures the compiler must handle two topics differently. First, the function returns with an `RTI`. Second, all modified registers must be saved. The processor D, X, and Y registers are saved by the hardware. The Compiler must additionally save the page registers if they are to be modified inside of the function.

#pragma TRAP_PROC

Which page registers are saved is determined by the `TRAP_PROC` pragma. The syntax of this pragma is

```
#pragma TRAP_PROC [SAVE_ALL_REGS | SAVE_NO_REGS]
```

If `TRAP_PROC SAVE_ALL_REGS` is used, all page registers are saved, whether or not they are used in the interrupt procedure. If `TRAP_PROC SAVE_NO_REGS` is given, no page registers are saved. If only `TRAP_PROC` is given, all page registers specified with the `-Cp` option are saved. It is up to you to ensure that no other page registers are modified.

NOTE The page registers are changed by paged data accesses. For details, see the Paged Variables section.

Interrupt Vector Table Allocation

The Compiler provides a non-ANSI compliant way to directly specify the interrupt vector number in the source:

```
void interrupt 0 ResetFunction(void) {
  /* reset handler */
}
```

The Compiler uses the following translation from interrupt vector number to interrupt vector address (Table 10.11).

Table 10.11 Vector relationships

Vector Number	Vector Address	Vector Address Size
0	0xFFFE, 0xFFFF	2
1	0xFFFC, 0xFFFD	2
2	0xFFFA, 0xFFFB	2

Table 10.11 Vector relationships

Vector Number	Vector Address	Vector Address Size
...	...	...
n	0xFFFF - (n*2)	2

Debug Information

The following debug information must be considered for the HC12 Compiler.

- There is no debug information for variables held in a register. This may happen if either register variables are enabled (`-Or` compiler option to switch on), or variables are allocated by the induction variable optimization (`-Ol0` compiler option to switch it off, it is enabled - the default). In addition, the `-Ou` compiler option removes stores to local variables when possible. The last parameter of a function is passed in a register if its size is smaller or equal to four bytes. When this parameter is accessed while it is still in the register at the start of a function, it is never stored to the stack. When a variable is never stored to the stack, no space is allocated for it and the debug information says that this variable is not allocated.
- The common code optimization does not generate any source positions inside common code. Some linear sequences may not contain any marker at all. Previous compiler versions did generate source position inside of common code. Then single stepping inside of such code did move the whole function. Seeing the source code, it is often not obvious which code is common code. The common-code optimization is switched off with `-Onf`.
- The `BRA` to `RET` peephole optimization (`-OnP=r`) and the `JSR/RTS` optimization causes the final `RTS` instruction at the end of a function to not always be executed. Setting a breakpoint at the last `RTS` will not always stop the application.
- The `JSR/RTS` peephole optimization removes the stack frame of a function from the stack before it is logically finished. Such functions disappear from the call chain. A step out from the last called function steps out two functions wide.
- The Debugger is not aware of constants in the code. Those constants may come from `DC` instructions (Assembler/HLI Assembler) or from tables used for switch processing. The disassembly module of the debugger tries to decode those constants as normal processor instructions.
- The Debugger is unaware of switch runtime routines. A step over a call of a switch runtime routine does not stop at the next statement. But source stepping works. When the runtime routine is found, the debugger will step in. When the runtime routine is finished, the debugger will continue at the right place. It is not recommended to use step over at the switch selector.
- The long-branch optimization replaces a long branch with a short one to a place which also branches to the same target. When debugging the intermediate branch instruction this also occurs, although there seems to be no relation to the code actually executed. Use `-OnB=l` to switch this optimization off.
- The short-branch optimization replaces a branch always over two bytes or one byte with the opcodes “`BNE`” or “`CPS #`”. In the second and third byte of this

instruction, other assembler instructions are encoded. This situation is not known by the decoder or the assembly window of the simulator/debugger. It seems that some branches are targeting inside of `BNE` and `CPS` instructions. Please see the manual for details about this optimization. Use `-OnB=a` to switch this optimization off.

- The HIWARE object file format and ELF/DWARF 1.1 do not support multiple C source files. When several source files in one compilation unit contain code, the debug information is correct only for the main file (the one noted on the command line). This problem arises from the fact that debug information formats do not support multiple source files. This is no limitation of the compiler/simulator/ debugger. ELF/DWARF 2.0 fully supports this situation, where correct debugging is also possible. Please note that source code in header files is the usual case for C++ inline functions.
- ELF/DWARF object files do not yet handle smart linking for data objects. Objects not linked by a smart linker just have address zero as debug information. In embedded applications, an object is placed at address zero so the debugger cannot detect that such an object has been removed. Therefore, it lists such an object as a normal object.

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 (Listing 10.16):

Listing 10.16 CODE_SEG and DATA_SEG pragmas

```
#pragma CODE_SEG [__NEAR_SEG|__FAR_SEG|__SHORT_SEG] <name>

#pragma DATA_SEG [__GPAGE_SEG__|__RPAGE_SEG| DPAGE_SEG|__PPAGE_SEG|
__EPAGE_SEG|__SHORT_SEG] <name>
```

Both are valid until the next pragma of the same type 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 current these default segments, use the segment name `DEFAULT`:

```
#pragma CODE_SEG DEFAULT
```

```
#pragma DATA_SEG DEFAULT
```

The additional `__SHORT_SEG` keyword informs the Compiler that a data segment is allocated in the zero page (address range from `0x0000` to `0x00FF`):

```
#pragma DATA_SEG __SHORT_SEG <segment_name>
```

or

```
#pragma DATA_SEG __SHORT_SEG DEFAULT
```

Using the zero page enables the Compiler to generate much denser code because the *DIRECT* addressing mode is used instead of *EXTENDED*.

NOTE It is the programmer's responsibility to actually allocate `__SHORT_SEG` segments in the zero page in the Linker parameter file. For more information, see the Linker section in the Build Tools manual.

The `__far` and `__near` keywords specify the calling convention for functions. `__far` function calls set the PPAGE register. `__near` function calls must stay in the same page. In the *BANKED* memory model, functions are `__far` - the default. In the *SMALL* memory model, functions are `__near` - the default case.

The `DPAGE`, `EPAGE`, and `PPAGE` keywords are used to specify the page register for paged variables. For details see Paged Variables.

Optimizations

The Compiler applies a variety of code improving techniques commonly defined as "optimizations". This section gives a short overview about the most important optimizations.

Lazy Instruction Selection

Lazy instruction selection is a very simple optimization that replaces certain instructions by shorter or faster equivalents. Examples are the use of `TSTA` instead of `CMPA    #0` or using `COMB` instead of `EORB    #0xFF`.

Peephole Optimizations

The peephole optimizer replaces longer code patterns with shorter ones. All peephole optimizations are switched off together with `-OnP` or each peephole optimization is switched off separately with the `-OnP={<char>}` command line option. Peephole optimizations are not done for inline assembler code.

LEAS to PUSH/POP Optimization (-OnP=a to disable it)

```
    LEAS -2,SP
```

is optimized to:

```
    PSHD
```

This optimizations uses `PULL` or `POP` for small SP changes instead of using LEAS. This optimization is switched off by the `-Ot` command line option, optimize for time.

POP PULL Optimization (-OnP=b to disable it)

```
    PSHA
    PULA
```

A value is pushed and immediately afterwards popped again, so both instructions are removed.

Compare 0 Optimization (-OnP=c to disable it)

```
L2: LDD   a
    CPD   #0
    BNE   L2
```

is optimized to:

```
L2: LDD   a
    BNE   L2
```

This optimization avoids compares to 0 if the flags are already set by another instruction.

Load/Store Optimization (-OnP=d to disable it)

```
    STD   a
    LDD   a
```

is optimized to:

```
    STD   a
```

This optimization removes redundant loads and stores. The load/store optimization traces the used registers and the memory. The optimization is only done if neither the registers are modified nor the memory is accessed.

LEA/LEA Optimization (-OnP=e to disable it)

This optimization does not work if there are instructions between the two LEAs (for that case, use the Load/Store optimization).

```
LEAX 2,X
LEAX 2,X
```

is optimized to:

```
LEAX 4,X
```

Load/Store to POP/PUSH Optimization (-OnP=f to disable it)

```
STD   2,-SP
```

is optimized to:

```
PSHD
```

Instead of creating `PULL` and `POP` instruction, the Compiler generates normal load and stores to the stack with explicit stack pointer changes. Such instructions can sometimes be combined with explicit stack pointer changes. Otherwise, the load and store operations are converted by peephole optimization into `PULL` and `POP` instructions.

Load Arithm Store Optimization (-OnP=g to disable it)

```
LDAA  c
INCA
STAA  c
```

is optimized to:

```
INC   c
```

and

```
LDAA  0,Y
ANDA  #0x0f
STAA  0,Y
```

is optimized to:

```
BCLR  0,Y,#240
LDAA  0,Y
```

JSR/RTS Optimization (-OnP=h to disable it)

```
    JSR function
    RTS
```

is optimized to:

```
    JMP function
```

NOTE This optimization removes stack frames before calling other functions. While debugging, this optimization removes functions from the call chain when the last function is called, but not when this function is actually finished. For better debug information, this optimization can selectively be switched off by using the `-OnP=h` option.

INC/DEC Compare Optimization (-OnP=i to disable it)

```
L3: ADDD  #1
    BNE   L3
```

is optimized to:

```
L3: IBNE  D,L3
```

Store/Store Optimization (-OnP=j to disable it)

```
    STD   b
    INCA
    STD   b
```

is optimized to:

```
    INCA
    STD b
```

The store/store optimization traces only the memory accesses. The optimization is done only if no memory access occurs between the two stores.

LEA 0 Optimization (-OnP=k to disable it)

```
    LEAS  0,SP
```

is optimized to:

```
/* no instruction */
```

LEA into Addressing Mode Optimization(-OnP=l to disable it)

```
LEAS  2,SP
STD   0,SP
```

is optimized to:

```
STD   2,+SP
```

and

```
LEAS  2,SP
STD   2,+SP
```

is optimized to:

```
STD  4,+SP
```

The compiles tries to move `LEAX`, `LEAY`, and LEAS instructions into register indirect memory accesses. The `LEA` into addressing mode optimization includes also an `LEA`/`LEA` optimization. The other `LEA`/`LEA` optimization does not handle instructions between the two `LEA`s.

```
LEAX  2,X
NOP
LEAX  2,X
```

is optimized to:

```
NOP
LEAX  4,X
```

RTS/RTS Optimization (-OnP=m to disable it)

```
RTS
RTS
```

is optimized to:

```
RTS
```

BCLR, BCLR Optimization (-OnP=n to disable it)

```
BCLR  0,Y, 0x01
BCLR  0,Y, 0x02
```

is optimized to:

```
BCLR  0,Y,#3
```

PULL POP Optimization (-OnP=p to disable it)

```
PULA
PSHA
CLRA
```

is optimized to:

```
CLRA
```

PSHC PULC optimization (-OnP=q to disable it)

With the -Or: Allocate Local Variables into Registers or -Ol: Try to Keep Loop Induction Variables in Registers compiler options, the compiler sometimes generates unnecessary `PSHC` and `PULC` instructions during code generation. When some stores, loads and transfers are done before the instruction sets some flags, `PSHC` and `PULC` are not necessary. The compiler does this in order for the peephole optimizer to remove them, wherever possible. This optimization actually improves intentionally generated code patterns. This optimization moves the loads, stores, and transfers and removes the `PSHC` and `PULC`, if possible.

```
LDAA  0,SP
PSHC
LDX   2,SP
PULC
```

is optimized to:

```
LDX   1,SP
LDAA  0,SP
```

BRA to RTS Optimization (-OnP=r to disable it)

```
      BRA lrts
     ...
lrts: RTS
```

is optimized to:

```
      RTS
      ...
lrts: RTS
```

Unconditional branches to an RTS are directly replaced with an RTS.

NOTE When debugging, it may happen that a function finishes although there is a breakpoint at the last instruction. This is avoided with this option.

TFR/TFR Optimization (-OnP=t to disable it)

```
    TFR   D,X
    TFR   D,X
```

is optimized to:

```
    TFR   D,X
```

Unused Optimization (-OnP=u to disable it)

```
    INCA
    CLRA
    STAA  a
```

is optimized to:

```
    CLRA
    STAA  a
```

Removing unnecessary compare instruction (-OnP=v to disable it)

This optimization removes unnecessary compare instructions in Listing 10.17:

Listing 10.17 Example of the "removing unnecessary-compare instruction" optimization

```
With -OnP=v:
   CPX <opr>
   BLE L1
   CPX <opr>  ; This is the unnecessary compare instruction.
   BNE L2
...

Without -OnP=v:
   CPX <opr>
   BLE L1
   BNE L2
   ...
```

The optimization may also be disabled by setting the '`volatile`' attribute for `<opr>`.

Peephole index optimization (-OnP=x to disable it)

This optimization uses the Accumulator-Offset Indexed Addressing mode (Listing 10.18) instead of using one of the Constant-Offset Indexed Addressing modes.

Listing 10.18 Example of peephole index optimization

```
unsigned char arr[12];
unsigned char index;
unsigned char test(void) {
  return arr[index];
}

With -OnP=x:
   LDAB  index
   CLRA
   TFR   D,X
   LDAB  arr,X
   RTS

Without -OnP=x:
   LDAB  index
   LDX   #arr
   LDAB  B,X
   RTS
```

Branch Optimizations

The Compiler uses branch instructions with 1-byte offsets whenever possible. In addition, other optimizations for branches are also available.

Short BRA Optimization (-OnB=a to disable it)

A branch over one byte is replaced with the opcode of "`BRN`". A branch over two bytes is replaced with the opcode of "`CPS      #`" (Listing 10.19).

Listing 10.19 Short BRA optimization example

```
int q(void) {
  if (f()) {
    return 1;
  } else {
    return 0;
  }
}
```

The code produced with this optimization:

```
0000 160000  JSR   f
0003 044403  TBEQ  D,3   ;abs = 0009
0006 C601    LDAB  #1
0008 21C7    BRN   -57   ;abs = FFD1
000A 87      CLRA
000B 3D      RTS
```

With the `-OnB=a (disable short BRA optimization)` option the Compiler produces one more byte:

```
0000 160000  JSR   f
0003 044404  TBEQ  D,4   ;abs = 000A
0006 C601    LDAB  #1
0008 2001    BRA   1     ;abs = 000B
000A C7      CLRB
000B 87      CLRA
000C 3D      RTS
```

The branch optimizer replaces the "BRA 1" in the second example with the opcode of "BRN", 0x21. Then the Decoder joins the BRN with the CLRB to one BRN. Actually the Decoder writes something like the following:

```
0008 21       "BRA 1"
000A C7       CLRB
```

The CLRB out of the second code disappears in the first listing into the offset of the BRN instruction. The same type of optimization is also done with a "BRA 2". Then the opcode of a "CPS #" is taken.

NOTE BRN and CPS in a Decoder listing are often the result of this optimization. If so, one or two additional machine instructions are hidden after the opcode. The compiler will write this as SKIP1 or SKIP2 pseudo opcode to the listing file.

Branch JSR to BSR Optimization (-OnB=b to disable it)

This optimization uses a BSR instead of a JSR, if the offset is small enough and known.

Long Branch Optimization (-OnB=l to disable it)

This optimization tries to replace a long branch with a short branch to another branch, which branches to the same target (Listing 10.20).

Listing 10.20 Long branch optimization example

```
     ...
     LBNE  l0
     ...
     LBNE  l0
     // more than 0x80 bytes of code
l0:  ...
```

This situation is recognized and replaced with the following:

```
     ...
     BNE   l1
     ...
l1:  LBNE  l0
     // more than 0x80 bytes of code
l0:  ...
```

Branch Tail Optimization (-OnB=t to disable it)

Branch tail merging removes common code if the common code patterns branch to the same place.

Constant Folding

Constant folding options only affect constant folding over statements. The constant folding inside of expressions is always done.

Volatile Objects

The Compiler does not do register tracing on volatile objects. Accesses to volatile objects are not eliminated. It also does not change word operations to byte operations on volatile objects (as it does for other memory accesses) when the option `-CVolWordAcc` is specified.

Programming Hints

The MC68HC(S)12 is an 8/16-bit processor not designed with high-level languages in mind. You must observe certain points in order for 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.

- Allocate frequently used static variables in the zero page using `__SHORT_SEG` segments.
- Use variables of type `char` if the value range is large enough for your purpose (`0` to `255` for `unsigned char`; `-128` to `127` for `signed char`).

Consider however that expressions containing both `char` and `int` variables usually are worse than equivalent expressions containing only `int` variables because the `char` variables have to be extended first. The same also holds for certain expressions on characters like:

```
char a, b, c, d;
a = (b + c) / d;
```

or

```
if (a+1 < b) ...
```

because they must be evaluated to 16 bits to comply to the semantics of ANSI–C.

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!
- Avoid stack frames larger than 256 bytes. The stack frame includes the parameters, local variables, and usually some additional bytes for temporary values.
- Avoid structs larger than 256 bytes if the fields are accessed via pointers.

11 High-Level Inline Assembler for the Freescale HC(S)12

The HLI (High Level Inline) Assembler provides a means to make full use of the properties of the target processor right 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 further information, please refer to the HC12 Reference Manual.

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 (Listing 11.1 through Listing 11.5.

Listing 11.1 Inline assembly - version #1

```
__asm <Assembly Instruction> ; [/* Comment */]
__asm <Assembly Instruction> ; [// Comment]
```

Listing 11.2 Inline assembly - version #2

```
__asm {
  { <Assembly Instruction> [; Comment] \n}
}
```

NOTE (In above syntax, the closing '}' has to be on a new line.

Listing 11.3 Inline assembly - version #3

```
__asm ( <Assembly Instruction> [; Comment] ) ;
```

Listing 11.4 Inline assembly - version #4

```
    __asm [(] <string Assembly instruction> [)] [;]

where the <string Assembly instruction> =
          <Assembly Instruction>  [; <Assembly instruction>]
```

Listing 11.5 Inline assembly - version #5

```
    #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.

Mixing HLI Assembly and HLL

Mixing High Level Inline (HLI) Assembly with a High Level Language (HLL, e.g., C or C++) requires special attention. The Compiler does not care about used or modified registers in HLI Assembly, thus you have save or restore registers which are used in HLI. This is not a problem if a function contains HLI Assembly only. It is recommended to place complex HLI Assembly code, or HLI Assembly code modifying any registers, into separate functions. See Listing 11.6 for a problematic case mixing C and HLI assembly.

Listing 11.6 Function whereby HLI assembly code modifies a register

```
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. If the register is modified in the HLI block, this may crash your code.

A simple example illustrates the use of the HLI-Assembler (Listing 11.7). 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 (we assume the *SMALL* memory model):

Listing 11.7 HLI Assembler example

```
#pragma NO_ENTRY
void strcpy (char *from, char *to)
  /* 'to' is passed in D
     'from' is passed on the stack SP:2 */
{
  __asm {
    TFR   D,X
    LDY   2,SP
  loop:
    LDAA  1,Y+
    STAA  1,X+
    BNE   loop
  }
}
```

NOTE If `#pragma NO_ENTRY` is not set, the Compiler takes care of entry and exit code. You do not have to worry about setting up a stack frame.

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 not allowed (Listing 11.8):

Listing 11.8 Unallowed C macro form

```
#define NOT_OK { __asm { \
                     NOP; \
                       NOP; \
                }
```

The NOT_OK macro is expanded by the preprocessor to one single line, which is then incorrectly translated because every assembly instruction must be explicitly terminated by a new line. Use #pragma NO_STRING_CONSTR: No String Concatenation during preprocessing to build immediates by using # inside macros.

Special Features

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 over function calls.

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 (LDAA, STX, ...)
- All register names (A, B, D, X, Y, CCR, SP)
- The identifier PAGE

For these reserved words, the inline assembler is *not* case-sensitive, i.e., LDAB is the same as ldab or even LdAb. For all other identifiers (labels, variable names, and so on) the inline assembler is case-sensitive.

Pseudo–Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are listed in Listing 11.9:

Listing 11.9 Pseudo opcodes for constants

```
DC.B  1     ; Byte constant 1
DC.B  0     ; Byte constant 0
DC.W  12    ; Word constant 12
DC.L  20,23 ; Longword constants
```

Accessing Variables

The inline assembler allows accessing local and global variables declared in C by using their names in the instruction. Global variable names are translated into the *EXTENDED* or *DIRECT* addressing mode, depending upon which segment the variable is located.

Constant Expressions

Constant expressions may be used anywhere an *IMMEDIATE* value is expected. They may contain the binary operators for addition (“+”), subtraction (“-”), multiplication (“*”), and division (“/”). Also, the unary operator “–” is allowed. Round brackets may be used to force an evaluation order other than the normal one. The syntax of numbers is the same as in ANSI-C.

NOTE You cannot use ’$’ for hexadecimal constants.

Addresses of Variables

A constant expression may also be the address of a global variable or the offset of a local variable.

```
AddrOfVar = "@|#"<Variable.
```

As examples:

```
LDX @g ; Load X with address of global variable
LDY #l ; Load Y with frame offset of local variable or
         parameter
```

For HCS12X devices the `@` and `#` operators generate relocations for the logical address space. If you want to have a relocation for the global address space you need to specify the name of the global relocation type (only the `#` operator is accepted for relocation specifications), i.e., `#GLOBAL`

```
LDX #GLOBAL(g) ; Load X with address of global variable
```

NOTE For HCS12X devices the `#LOGICAL` operator is the same as the `#` operator.

It is also possible to access the fields of a `struct` or a `union` by using the normal ANSI-C notation.

```
LDD   r.f      ; Load D with content of field f.
```

The inline assembler enables you to specify an offset from the address of a variable in order to access the low word of a long or a float variable:

```
Offset   = ":" ConstExpr.
Variable = Ident {"." Ident}.
```

Below are some examples (assuming all variables are long):

```
LDY   @g:2  ; Load Y with ((address of g) + 2)
LDX   g:2   ; Load X with the value stored there
LDD   r.f:2 ; Load D with low word of field f.
```

This feature may also be used to access array elements with a constant index:

```
int   a[20] ;
LDD   a:24  ; Load a[12] into D
```

In the *BANKED* memory model, it is sometimes necessary to specify the bank number of the memory bank where a particular function is allocated. This can be done with the `#PAGE` relocation operator:

```
LDAB #PAGE(g); Load B with page address of global variable
```

For the HCS12X devices `#PAGE` generates a page relocation for the logical address space. If you want to have a page relocation for the global address space you need to specify `#GLOBAL_PAGE:`

```
LDAB #GLOBAL_PAGE(g); Load B with global page address of
                      global variable
```

NOTE For HCS12X devices the `#LOGICAL_PAGE` operator is the same as the `#PAGE` operator.

III

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 the Standard Library: Discussion of all types and macros defined in the ANSI-C standard library.
- The Standard Functions: Description of all functions in the ANSI-C library

12 Library Files

Directory Structure

The library files are delivered in the following structure (:Listing 12.1).

Listing 12.1 Layout of files after a CodeWarrior installation/

```
<install>\lib\<target>c\         /* readme files, make files */
<install>\lib\<target>c\src      /* C library source files   */
<install>\lib\<target>c\include  /* library include files    */
<install>\lib\<tartet>c\lib      /* default library files    */
<install>\lib\<target>c\prm      /* Linker parameter files   */
```

Check out the `README.TXT` located in the library folder with additional information on memory models and library filenames.

How to Generate 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 CodeWarrior and build all the targets.

Common Source Files

Table 12.1 lists the source and header files of the Standard ANSI Library that are not target-dependent.

Table 12.1 Standard ANSI Library—Target Independent Source and Header Files

Source File	Header File
alloc.c	
assert.c	assert.h

Table 12.1 Standard ANSI Library—Target Independent Source and Header Files

Source File	Header File
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

Target Dependent Files for HC12

Table 12.2 lists the target dependent Standard ANSI Library files.

Table 12.2 Standard ANSI Library—Target Dependent Source and Header Files

Source File	Header File	Description
	default.sgm	Segment file
	hidef.h	HI-CROSS+ specific definitions
	math.h	part of ANSI library
	non_bank.sgm	Segment file
setjmp.c	setjmp.h	part of ANSI library
signal.c		part of ANSI library
start12.c	start12.h	Startup

Table 12.2 Standard ANSI Library—Target Dependent Source and Header Files

Source File	Header File	Description
	system.h	Runtime prototypes
dadd.c		part of runtime support (IEEE64)
dansi.c		part of runtime support (IEEE64)
datapage.c		part of runtime support (far pointers)
dcmp.c		part of runtime support (IEEE64)
dconv.c	dconf.h	part of runtime support (IEEE64)
dconv.c	dconv.h	part of runtime support (IEEE64)
dmul.c		part of runtime support (IEEE64)
dregs.c	dregs.h	part of runtime support (IEEE64)
fadd.c		part of runtime support (IEEE32)
fansi.c		part of runtime support (IEEE32)
fcmp.c		part of runtime support (IEEE32)
fconv.c		part of runtime support (IEEE32)
fmul.c		part of runtime support (IEEE32)
fregs.c	fregs.h	part of runtime support (IEEE32)
rtshc12.c		part of runtime support (integer, long, switches)
	runtime.sgm	Segment declaration for runtime functions
vregs.c	vregs.h	part of runtime support

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 has to 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
- Register initialization
- Handling ROM libraries

Because not all of the above tasks may be needed for an application and for efficiency reasons, special startup is provided as well (e.g., written in HLI). However, 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.

Startup Files for the Freescale HC12

To initialize global variables either a pre-built startup object file has to be linked or the start12.c source file has to be compiled with your project. Adding start12.c is recommended as the correct setup is automatically detected at compile time.

Depending on the memory model, a different startup object file has to be linked to the application. See Table 12.3

Table 12.3 Startup Object File Required by Each Memory Model

Startup Object File	Core	Memory Model	Source File	Compiler Options
start12s.o	HC12/ HCS12	Small	start12.c	-Ms
start12b.o	HC12/ HCS12	Banked	start12.c	-Mb
start12l.o	HC12/ HCS12	Large	start12.c	-Ml
strt12sp.o	HC12/ HCS12	Small (1)	start12.c	-Ms -C++f
strt12bp.o	HC12/ HCS12	Banked (1)	start12.c	-Mb -C++f
strt12lp.o	HC12/ HCS12	Large (1)	start12.c	-Ml -C++f
start12xs.o	HCS12X	Small	start12.c	-Ms -CpuHCS12X
start12xb.o	HCS12X	Banked	start12.c	-Mb -CpuHCS12X

Table 12.3 Startup Object File Required by Each Memory Model

Startup Object File	Core	Memory Model	Source File	Compiler Options
start12xl.o	HCS12X	Large	start12.c	-Ml -CpuHCS12X
start12xsp.o	HCS12X	Small (1)	start12.c	-Ms -C++f -CpuHCS12X
start12xbp.o	HCS12X	Banked (1)	start12.c	-Mb -C++f -CpuHCS12X
start12xlp.o	HCS12X	Large (1)	start12.c	-Ml -C++f -CpuHCS12X

(1): C++ global constructors are called

Library Files

Most of the object files of the ANSI library are delivered in the form of an object library (see below).

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.

Please 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.

13

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 reentrant, 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.

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 where to put the heap, how big should it be, and what should happen when the heap memory runs out.

All these points can be solved in the `"heap.c"` file. The heap simply is viewed as a large array, and there is a default error handling function. Feel free to 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`.

Signals - signal.c

Signals have been implemented in a very rudimentary way - as traps. This means, the signal() function allows you to set a vector to some function of your own (which of course should be a `TRAP_PROC`), while the raise() function is not implemented. If you decide to ignore a certain signal, a default handler is installed that does nothing.

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 have not been implemented. If these functions are needed, the programmer will have to specifically write them.

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() simply perform a `HALT`.

I/O - printf.c

The printf() library function is not implemented 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 13.1).

Listing 13.1 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 several disadvantages of 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 13.2):

Listing 13.2 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 Listing 13.3:

Listing 13.3 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()`.

Locales - locale.*

Has not been implemented.

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.

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 it is recommended to switch off this macro (set `ENABLE_OVERFLOW_CHECK` to 0).

14

Library Structure

In this section, 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.

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 14.1):

Listing 14.1 Error handling functions

```
void  assert(int expr);
void  perror(const char *msg);
char * strerror(int errno);
```

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 14.2).

Listing 14.2 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);
```

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 14.3).

Listing 14.3 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);
```

Mathematical Functions

The ANSI library contains a variety of floating point functions. The standard interface, which is defined for type `double` (Listing 14.4), has been augmented by an alternate interface (and implementation) using type `float`.

Listing 14.4 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 14.5).

Listing 14.5 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 14.6):

Listing 14.6 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 (Listing 14.7) and a function for generating a seed to start the random-number generator:

Listing 14.7 Random number generator functions

```
int  rand(void);
void srand(unsigned int seed);
```

Memory Management

To allocate and deallocate memory blocks, the ANSI library provides the following functions (Listing 14.8):

Listing 14.8 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.c file,` which can be modified by the user to fit a particular memory layout.

Searching and Sorting

The ANSI library contains both a generalized searching and a generalized sorting procedure (Listing 14.9):

Listing 14.9 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 14.10 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 14.11).

Listing 14.11 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);
```

System Functions

The ANSI standard includes some system functions for raising and responding to signals, non–local jumping, and so on.

Listing 14.12 ANSI-C system functions

```
void       abort(void);
int        atexit(void(* func) (void));
void       exit(int status);
char*      getenv(const char* name);
int        system(const char* cmd);
int        setjmp(jmp_buf env);
void       longjmp(jmp_buf env, int val);
_sig_func signal(int sig, _sig_func handler);
int        raise(int sig);
```

To process variable-length argument lists, the ANSI library provides the following “functions” (Listing 14.13). (They are implemented as macros):

Listing 14.13 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);
```

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 14.14).

Listing 14.14 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);
```

Locale Functions

These functions are for handling locales. The ANSI–C library only supports the minimal "C" environment (Listing 14.15).

Listing 14.15 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);
```

Conversion Functions

Functions for converting strings to numbers are found in Listing 14.16.

Listing 14.16 ANSI-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);
```

printf() and scanf()

More conversions are possible for the C functions for reading and writing formatted data. These functions are shown in Listing 14.17.

Listing 14.17 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, ...);
```

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, is therefore impossible for Freescale to provide an implementation for these features that the user has to specifically implement.

Listing 14.18 contains file I/O functions while Listing 14.19 has functions for the reading and writing of characters. The functions for reading and writing blocks of data are found in Listing 14.20. Functions for formatted I/O on files are found in Listing 14.21, and Listing 14.22 has functions for positioning data within files.

Listing 14.18 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 14.19 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 14.20 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 14.21 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 14.22 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(
```

15 Types and Macros in the Standard Library

This section discusses all types and macros defined in the ANSI standard library. We cover each of the header files, in alphabetical order.

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
```

float.h

Defines constants describing the properties of floating point arithmetic. See Table 15.1 and Table 15.2.

Table 15.1 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 `long double`.

Table 15.2 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 $\text{FLT\_RADIX}^{\text{MAX\_EXP}}$ is a finite normalized value.
MAX_10_EXP	Maximum exponent such that $10^{\text{MAX\_10\_EXP}}$ is a finite normalized value.
MIN	Smallest positive normalized value.
MIN_EXP	Smallest negative exponent such that $\text{FLT\_RADIX}^{\text{MIN\_EXP}}$ is a normalized value.
MIN_10_EXP	Smallest negative exponent such that $10^{\text{MIN\_10\_EXP}}$ is a normalized value.

limits.h

Defines a couple of constants for the maximum and minimum values that are allowed for certain types. See Table 15.3.

Table 15.3 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

Table 15.3 Constants Defined in limits.h (*continued*)

Constant	Description
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
LONG_MIN	Minimum value for long int
LONG_MAX	Maximum value for long int
ULONG_MAX	Maximum value for unsigned long int

locale.h

The header file in Listing 15.1 defines a struct containing all the locale specific values.

Listing 15.1 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 should be set. See Table 15.4.

Table 15.4 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.

math.h

Defines just this constant:

`HUGE_VAL`

Large value that is returned if overflow occurs.

setjmp.h

Contains just this type definition:

```
typedef jmp_buf;
```

A buffer for setjmp() to store the current program state.

signal.h

Defines signal handling constants and types. See Table 15.5 and Table 15.6.

```
typedef sig_atomic_t;
```

Table 15.5 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.

Signal Type Constants. (Table 15.6).

Table 15.6 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

stddef.h

Defines a few generally useful types and constants. See Table 15.7.

Table 15.7 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.

stdio.h

There are two type declarations in this header file. See Table 15.8.

Table 15.8 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().

Table 15.9 lists the constants defined in stdio.h.

Table 15.9 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().

Table 15.9 Constants defined in stdio.h (*continued*)

Constant	Description
_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;
```

stdlib.h

Besides a redefinition of `NULL`, `size_t` and `wchar_t`, this header file contains the type definitions listed in Table 15.10.

Table 15.10 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().

Table 15.11 lists the constants defined in stdlib.h

Table 15.11 Constants Defined in stdlib.h

Constant	Definition
EXIT_FAILURE	Exit code for unsuccessful termination.
EXIT_SUCCESS	Exit code for successful termination.

Table 15.11 Constants Defined in stdlib.h (*continued*)

Constant	Definition
RAND_MAX	Maximum return value of rand().
MB_LEN_MAX	Maximum number of bytes in a multi-byte character.

time.h

This header files defines types and constants for time management. See Listing 15.2.

Listing 15.2 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.

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.

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 Listing 15.3.

Listing 15.3 Use assert() to assist in debugging

```
#ifdef NDEBUG
  #define assert(EX)
#else
  #define assert(EX) ((EX) ? 0 : _assert(__LINE__, __FILE__))
#endif
```

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 Listing 15.4 for an example using `stdarg.h`

Listing 15.4 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)`" would not be correct unless `res` would have the type `int` and not `char`.

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 Listing 15.5 are explained below together with all other ANSI functions.

Listing 15.5 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__ */
```

16

The Standard Functions

This section describes all the standard functions in the ANSI–C library. Each function description contains the subsections listed in Table 16.1.

Table 16.1 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.

Functions not implemented because the implementation would be hardware-specific anyway (e.g., `clock()`) are marked by:

Hardware specific

appearing in the right margin next to the function’s name. Functions for file I/O, which also depend on the particular hardware’s setup and therefore also are not implemented, are marked by:

File I/O

in the right margin.

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()

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()

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()

asctime()

Hardware specific

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()

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()

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()

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()

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()

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()

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()

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()

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()

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 should return an integer according to (Table 16.2):

Table 16.2 Return value from the comparison function, cmp_func()

If the key element is...	the return value should be...
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 (Table 16.3) of `bsearch()` are:

Table 16.3 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`.

calloc()

Hardware specific

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()`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

Return

`calloc()` returns a pointer to the allocated memory block. If the block could not be allocated, the return value is `NULL`.

See also

malloc() and

realloc()

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()

clearerr()

File I/O

Syntax

```
#include <stdio.h>

void clearerr(FILE *f);
```

Description

`clearerr()` resets the error flag and the `EOF` marker of file `f`.

clock()

Hardware specific

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()

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`. `x` should be expressed 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()

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()

ctime()

Hardware specific

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()

difftime()

Hardware specific

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()

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()

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()

exp() and expf()

Syntax

```
#include <math.h>

double exp (double x);
float  expf(float x);
```

Description

`exp()` computes e^x, where `e` is the base of natural logarithms.

Return

e^x. 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()

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()

fclose()

File I/O

Syntax

```
#include <stdlib.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()

feof()

File I/O

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.

ferror()

File I/O

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.

fflush()

File I/O

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()

fgetc()

File I/O

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()

fgetpos()

File I/O

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()

fgets()

File I/O

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()

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()

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()

fopen()

File I/O

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. Table 16.4 lists all possible modes.

Table 16.4 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.
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()

fprintf()

Syntax

```
#include <stdio.h>

int fprintf(FILE *f, const char *format, ...);
```

Description

`fprintf()` is the same as `sprintf()`, but the output goes to file `f` instead of a string.

For a detailed format description see `sprintf()`.

Return

The number of characters written. If some error occurred, `EOF` is returned.

See also

printf() and

`vfprintf()`, `vprintf()`, and `vsprintf()`

fputc()

File I/O

Syntax

```
#include <stdio.h>

int fputc(int ch, FILE *f);
```

Description

`fputc()` writes a character to file `f`.

Return

The integer value of `ch`. If an error occurred, `fputc()` returns `EOF`.

See also

fputs()

fputs()

File I/O

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()

fread()

File I/O

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()

free()

Hardware specific

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. The default implementation is not reentrant and should therefore not be used in interrupt routines.

freopen()

File I/O

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()

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()

fscanf()

File I/O

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()

fseek()

File I/O

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 Table 16.5.

Table 16.5 Offset position into the file for the fseek() function

mode	Position is set to...
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()

fsetpos()

File I/O

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()

ftell()

File I/O

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, this value should not be used except as argument to `fseek()`.

Return

`-1`, if an error occurred; otherwise the current file position.

See also

fgetpos() and

fsetpos()

fwrite()

File I/O

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()

getc()

File I/O

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.

getchar()

File I/O

Syntax

```
#include <stdio.h>

int getchar(void);
```

Description

`getchar()` is the same as `getc()` (`stdin`). See `fgetc()` for more information.

getenv()

File I/O

Syntax

```
#include <stdio.h>

char *getenv(const char *name);
```

Description

`getenv()` returns the value of environment variable `name`.

Return

`NULL`

gets()

File I/O

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()

gmtime()

Hardware specific

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()

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. Table 16.6 describes the character ranges tested by the functions.

Table 16.6 Appropriate character range for the testing functions

Function	Tests whether ch is in the range...
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 ('!'-'~').
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()

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()

ldexp() and ldexpf()

Syntax

```
#include <math.h>

double ldexp (double x, int exp);
float  ldexpf(float x, int exp);
```

Description

`ldexp()` multiplies `x` by 2^{exp}.

Return

`x * ` 2^{exp}. 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()

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()

localeconv()

Hardware specific

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()

localtime()

Hardware specific

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()

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()

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`.

See also

exp() and expf() and

log10() and log10f()

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()

malloc()

Hardware specific

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()`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

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()

mblen()

Hardware specific

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()

mbstowcs()

Hardware specific

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()

mbtowc()

Hardware specific

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()

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()

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()

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()

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()`

mktime()

Hardware specific

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()

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()

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()

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., x^y.

Return

x^y,	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()

printf()

File I/O

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()

putc()

File I/O

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, you should make sure that `f` is not an expression having side effects! See `fputc()` for more information.

putchar()

File I/O

Syntax

```
#include <stdio.h>

int putchar(char ch);
```

Description

`putchar(ch)` is the same as `putc(ch, stdin)`. See `fputc()` for more information.

puts()

File I/O

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()

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 should return an integer according to Table 16.7.

Table 16.7 Return value from the comparison function, cmp_func()

If the key element is...	The return value should be...
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 Table 16.8.

Table 16.8 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.

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()

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`.

realloc()

Hardware specific

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`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

Return

`realloc()` returns a pointer to the new memory block. If the operation could not be performed, the return value is `NULL`.

See also

free()

remove()

File I/O

Syntax

```
#include <stdio.h>

int remove(const char *filename);
```

Description

`remove()` deletes the file `filename`. If the file is open, `remove()` does not delete it and returns unsuccessfully.

Return

Non–zero, if there was an error; zero otherwise.

See also

tmpfile() and

tmpnam()

rename()

File I/O

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()

rewind()

File I/O

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()

scanf()

File I/O

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()

setbuf()

File I/O

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 (`buf` should point to an array of `BUFSIZ` bytes).

See also

fflush() and

setvbuf()

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()

setlocale()

Hardware specific

Syntax

```
#include <locale.h>

char *setlocale(int class, const char *loc);
```

Description

`setlocale()` changes the program's locale – either all or just part of it, depending on `class`. The new locale is given by the character string `loc`. The classes allowed are given by Table 16.9.

Table 16.9 Allowable classes for the setlocale() function

Class	Changes the locale...
LC_ALL	for all classes.
LC_COLLATE	for the strcoll() and strxfrm() functions.
LC_MONETARY	for monetary formatting.
LC_NUMERIC	for numeric formatting.
LC_TIME	for the strftime() function.
LC_TYPE	for character handling and multi–byte character functions.

CodeWarrior 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()

setvbuf()

File I/O

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 16.10 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()

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 Table 16.11.

Table 16.11 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 CodeWarrior, 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()

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()

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()

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 (Table 16.12).

Table 16.12 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.

Table 16.12 Effect of # in the Format specification (*continued*)

Conversion	Effect of "#"
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 `0`s 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 (Table 16.13).

Table 16.13 Effect of the Precision specification

Conversion	Precision
d, i, o, u, x, X	The minimum number of digits to print.
e, E, f	The number of fractional digits to print.
g, G	The maximum number of significant digits to print.
s	The maximum number of characters to print.
others	undefined.

If the Precision specifier is `"*"`, the precision is taken from the next argument, which must be an `int`. If that value is negative, the precision is ignored.

- `Length     = h|l|L`

A length specifier tells `sprintf()` what type the argument has. The first two length specifiers can be used in connection with all conversion characters for integral numbers. "`h`" defines `short`; "`l`" defines `long`. 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 (Table 16.14):

Table 16.14 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.
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 long int.
o	The argument, which must be an unsigned int; is printed in octal notation.

Table 16.14 Meaning of the Conversion Characters (*continued*)

Conversion	Description
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. Should only be given 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 which should be set if no floating point support is used. Some targets contain special libraries without floating point support.
The IEEE64 floating point implementation only supports printing numbers with up to 9 decimal digits. This limitation occurs because the implementation is using unsigned long internally which cannot hold more digits. Supporting more digits would increase the `printf()` size still more and would also cause the application to run considerably slower.

Return

The number of characters written to `s`.

See also

sscanf()

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()

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()

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 Table 16.15.

Table 16.15 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 Table 16.16.

Table 16.16 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 `short int` or a `long int`.

Table 16.16 Description of the action taken for each conversion.

Conversion	Description
`o`	Reads an octal number (syntax below). The parameter must be a pointer to an integral type.
`p`	Reads a pointer in the same format as `sprintf()` prints it. The parameter must be a `void **`.
`s`	Reads a character string up to the next white space character or at most `width` characters. The string is zero–terminated. The argument must be of type `char *`.
`u`	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. Should only be given 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:

```
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!

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()

strchr()

Syntax

```
#include <string.h>

char *strchr(const char *p, int ch);
```

Description

`strchr()` looks for character `ch` in string `p`. If `ch` is '`\0`', the function looks for the end of the string.

Return

A pointer to the character, if found; if there is no such character in `*p`, `NULL` is returned.

See also

memchr(),

strrchr(), and

strstr()

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()

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()

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()

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()

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()

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 (Table 16.17):

Table 16.17 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”.
%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.

Table 16.17 strftime() output string content and format (*continued*)

Format	Replaced with
%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., “1993”).
%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()

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.

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()

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()

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()

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()

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()

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()

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()

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()

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()

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:

```
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()

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()

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()

system()

Hardware specific

Syntax

```
#include <string.h>

int system(const char *cmd);
```

Description

`system()` executes the `cmd` command line

Return

Zero

tan() and tanf()

Syntax

```
#include <math.h>

double tan(double x);
float  tanf(float x);
```

Description

`tan()` computes the tangent of `x`. `x` should be 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()

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()

time()

Hardware specific

Syntax

```
#include <time.h>

time_t time(time_t *timer);
```

Description

`time()` gets the current calendar time. If `timer` is not `NULL`, it is assigned to it.

Return

The current calendar time.

See also

clock(),

mktime(), and

strftime()

tmpfile()

File I/O

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()

tmpnam()

File I/O

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.

See also

tmpfile()

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).

See also

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), and isxdigit(),

`toupper()`

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()

ungetc()

File I/O

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()

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. Listing 16.1 shows how to do it:

Listing 16.1 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);
}
```

vfprintf(), vprintf(), and vsprintf()

File I/O

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()

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()

wcstombs()

Hardware specific

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()

IV

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.

A

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

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.

Porting from Cosmic

If your current application is written for Cosmic compilers, there are some special things to consider.

How to Get Started...

The best way is if you create a new project using the New Project Wizard (in the CodeWarrior IDE: Menu File > New) or a project from a stationery template. This will set 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.

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.

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 (Listing A.1). The `equ` directive is not supported in normal C code.

Listing A.1 An example using the EQU directive

```
#ifdef  __MWERKS__
#define CLKSRC_B   0x00 /*; Clock source */
#else
  CLKSRC_B : equ   $00    ; Clock source
#endif
```

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 Listing A.2.

Listing A.2 Using a conditional block to account for different compilers

```
#ifdef __MWERKS__
  ldx @myVariable,x
  jsr MyFunction
#else
  ldx _myVariable,x
  jsr _MyFunction
#endif
```

may be really painful. Using macros which deal with the cases below (Listing A.3) is a better way to deal with this.

Listing A.3 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
```

so the source can use the macros:

`ldx USCRA(myVariable),x`

`jsr USCR(MyFunction)`

Pragma Sections

Cosmic uses the `#pragma section` syntax, while this compiler employs either `#pragma DATA_SEG` (Listing A.4) or `#pragma CONST_SEG` (Listing A.5).

or another example (for the data section):

Listing A.4 #pragma DATA_SEG

```
#ifdef __MWERKS__
#pragma DATA_SEG APPLDATA_SEG
#else
#pragma section {APPLDATA}
#endif
```

Listing A.5 #pragma CONST_SEG

```
#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.

Inline Assembly Constants

Cosmic uses an assembly constant syntax, whereas this compiler employs the normal C constant syntax (Listing A.6):

Listing A.6 Normal C constant syntax

```
#ifdef __MWERKS__
  and 0xF8
#else
  and #$F8
#endif
```

Inline Assembly and Index Calculation

Cosmic uses the + operator to calculate offsets into arrays. For CodeWarrior, you have to use a colon (:) instead:

Listing A.7 Using a colon for offset

```
ldx array:7
#else
  ldx array+7
#endif
```

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.

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 A.8)

Listing A.8 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
```

@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 A.9 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) { ....
```

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 A.10).

Listing A.10 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 A.11)

Listing A.11 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
}
```

Compiler Warnings

Check carefully the warnings produced by the compiler. 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 they are OK (e.g., using the `-W2: No Information and Warning Messages` option or using #pragma MESSAGE: Message Setting in the source code).

Linker *.lcf File (for the Cosmic compiler) and Linker *.prm File (for this compiler)

Cosmic uses a *.lcf 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
```

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__`

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.

@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);
```

@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;
```

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[];
```

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.

_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;`

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.

Interrupt Function, @interrupt

Interrupt functions have to be marked with #pragma TRAP_PROC or using the interrupt keyword (Listing A.12).

Listing A.12 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 */
}
```

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

Defining an Interrupt Function

The compiler provides two ways to define an interrupt function:

- Using pragma TRAP_PROC.
- Using the keyword interrupt.

Using the “TRAP_PROC” Pragma

The TRAP_PROC pragma informs the compiler that the following function is an interrupt function (Listing A.13). In that case, the compiler should terminate the function by a special interrupt return sequence (for many processors, an RTI instead of an RTS).

Listing A.13 Example of using the TRAP_PROC pragma

```
#pragma TRAP_PROC
void INCcount(void) {
  tcount++;
}
```

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 A.14).

Listing A.14 Example of using the “interrupt” keyword

```
interrupt void INCcount(void) {
  tcount++;
}
```

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.

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 A.15).

Listing A.15 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 A.16).

Listing A.16 Using the VECTOR command

```
VECTOR 69 INCcount
```

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 69, define the function as in Listing A.17.

Listing A.17 Definition of the INCcount() interrupt function

```
interrupt 69 void INCcount(void) {
int card1;
tcount++;
}
```

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.

Defining a Function in a Specific Segment

In order to define a function in a specific segment, use the CODE_SEG pragma (Listing A.18).

Listing A.18 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.*/
```

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 A.19).

Listing A.19 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
```

How to Use Variables in EEPROM

Placing variables into EEPROM is not explicitly supported in the C language. However, because EEPROM is widely available in embedded processors, a development tool for Embedded Systems must support it.

The examples are processor-specific. However, it is very easy to adapt them for any other processor.

Linker Parameter File

You have to define your RAM or ROM areas in your linker parameter file (Listing A.20). However, you should declare the EEROM memory as NO_INIT to avoid initializing the memory range during normal startup.

Listing A.20 Linker Parameter File

```
LINK test.abs

NAMES test.o startup.o ansi.lib END
SECTIONS
  MY_RAM = READ_WRITE 0x800 TO 0x801;
  MY_ROM = READ_ONLY  0x810 TO 0xAFF;
  MY_STK = READ_WRITE 0xB00 TO 0xBFF;
  EEPROM = NO_INIT    0xD00 TO 0xD01;
PLACEMENT
  DEFAULT_ROM  INTO  MY_ROM;
  DEFAULT_RAM  INTO  MY_RAM;
  SSTACK       INTO  MY_STK;
  EEPROM_DATA  INTO  EEPROM;
END
/* set reset vector to the _Startup function defined in startup code */
VECTOR ADDRESS 0xFFFE _Startup
```

The Application

The example in Listing A.21 shows an example which erases or writes an EEPROM word. The example is specific to the processor used, but it is easy to adapt if you consult the technical documentation about the EEPROM used for your derivative or CPU.

NOTE There are only a limited number of write operations guaranteed for EEPROMs so avoid writing to an EEPROM cell too frequently.

Listing A.21 Erasing and Writing an EEPROM

```
/*
 Definition of a variable in EEPROM.

   The variable VAR is located in EEPROM.
   - It is defined in a user-defined segment EEPROM_DATA
   - In the PRM file, EEPROM_DATA is placed at address 0xD00.

 Be careful, the EEPROM can only be written a limited number of times.
Running this application too frequently may surpass this limit and the
EEPROM may be unusable afterwards.
*/
#include <hidef.h>
#include <stdio.h>
#include <math.h>
/* INIT register. */
typedef struct {
  union {
    struct {
      unsigned int   bit0:1;
      unsigned int   bit1:1;
      unsigned int   bit2:1;
      unsigned int   bit3:1;
      unsigned int   bit4:1;
      unsigned int   bit5:1;
      unsigned int   bit6:1;
      unsigned int   bit7:1;
    } INITEE_Bits;
    unsigned char INITEE_Byte;
  } INITEE;
} INIT;
volatile INIT INITEE @0x0012;
#define EEON INITEE.INITEE.INITEE_Bits.bit0
/* EEPROG register. */
volatile struct {
  unsigned int   EEPGM:1;
  unsigned int   EELAT:1;
  unsigned int   ERASE:1;
  unsigned int   ROW:1;
  unsigned int   BYTE:1;
  unsigned int   dummy1:1;
  unsigned int   dummy2:1;
  unsigned int   BULKP:1;
} EEPROG @0x00F3;
/* EEPROT register. */
volatile struct {
  unsigned int   BPROT0:1;
  unsigned int   BPROT1:1;
  unsigned int   BPROT2:1;
  unsigned int   BPROT3:1;
  unsigned int   BPROT4:1;
  unsigned int   dummy1:1;
  unsigned int   dummy2:1;
  unsigned int   dummy3:1;
} EEPROT @0x00F1;
#pragma DATA_SEG EEPROM_DATA
unsigned int VAR;
#pragma DATA_SEG DEFAULT
void EraseEEPROM(void) {
  /* Function used to erase one word in the EEPROM. */
  unsigned long int i;
  EEPROG.BYTE = 1;
  EEPROG.ERASE = 1;
  EEPROG.EELAT = 1;
  VAR = 0;
  EEPROG.EEPGM =1;
  for (i = 0; i<4000; i++) {
    /* Wait until EEPROM is erased. */
  }
  EEPROG.EEPGM = 0;
  EEPROG.EELAT = 0;
  EEPROG.ERASE = 0;
}

void WriteEEPROM(unsigned int val) {
  /* Function used to write one word in the EEPROM. */
  unsigned long int i;
  EraseEEPROM();
  EEPROG.ERASE = 0;
  EEPROG.EELAT = 1;
  VAR = val;
  EEPROG.EEPGM = 1;
  for (i = 0; i<4000; i++) {
  /* Wait until EEPROM is written. */
  }
  EEPROG.EEPGM = 0;
  EEPROG.EELAT = 0;
  EEPROG.ERASE = 0;
}

void func1(void) {
  unsigned int i;
  unsigned long int ll;
  i = 0;
    do
    {
    i++;
    WriteEEPROM(i);
    for (ll = 0; ll<200000; ll++) {
    }
  }
  while (1);

}

void main(void) {
  EEPROT.BPROT4 = 0;
  EEON=1;
  WriteEEPROM(0);
  func1();
}
```

General Optimization Hints

Here are some hints how to reduce the size of your application:

- Check 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 both, you may remove the complete startup code and directly set up your stack in your 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. You can try -OdocF=`"-or"`. Using the -Li: List of Included Files option to write a log file displays the statistics for each single option.
- Check if you can use both IEEE32 for 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., 16 bits instead of 32 bits).
- Have a look into the map file to check runtime routines, which usually have a '_' prefix. Check for 32-bit integral routines (e.g., _LADD). 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 Search Switch Tables for an example).
- Finally, the linker has an option to overlap ROM areas (see the -COCC option in the linker).

Executing an Application from RAM

For performance reasons, it may be interesting to copy an application from ROM to RAM and to execute it from RAM. This can be achieved following the procedure below.

1. Link your application with code located in RAM.
2. Generate an S-Record File.
3. Modify the startup code to copy the application code.
4. Link the application with the S-Record File previously generated.

Each step is described in the following sections. The `fibo.abs` application is used for an example.

Link your application with code located in RAM.

We recommend that you generate a ROM library for your application. This allows you to easily debug your final application (including the copying of the code).

ROM Library Startup File

A ROM Library requires a very simple startup file, containing only the definition from the startup structure. Usually a ROM library startup file looks as follows:

```
#include "startup.h"

/* read-only: _startupData is allocated in ROM and ROM
Library PRM File */
struct _tagStartup _startupData;
```

You must generate a PRM file to set where the code is placed in RAM. As the compiler generates absolute code, the linker should know the final location of the code in order to generate correct code for the function call.

In addition, specify the name of the application entry points in the ENTRIES block of the PRM file. The application's main function, as well as the function associated with an Interrupt vector, must be specified there.

Suppose you want to copy and execute your code at address 0x7000. Your PRM file will look as in Listing A.22.

Listing A.22 Linker Parameter File

```
LINK fiboram.abs AS ROM_LIB
NAMES  myFibo.o start.o
END

SECTIONS
  MY_RAM = READ_WRITE 0x4000 TO 0x43FF;
  MY_ROM = READ_ONLY  0x7000 TO 0xBFFF; /* Dest. Address in RAM area */
PLACEMENT
  DEFAULT_ROM, ROM_VAR, STRINGS  INTO  MY_ROM;
  DEFAULT_RAM                    INTO   MY_RAM;
END
ENTRIES
  myMain
END
```

NOTE You cannot use a main function in a ROM library. Please use another name for the application's entry point. In the example above, we have used "`myMain`".

Generate an S-Record File

An S-Record File must be generated for the application. In this purpose, you can use the Burner utility.

The file is generated when you click the '1st byte(msb)' button in the burner dialog.

NOTE Initialize the field 'From' with 0 and the field 'Length' with a value bigger than the last byte used for the code. If byte `0xFFFF` is used, then Length must be at least `10000`.

Modify the Startup Code

The startup code of the final application must be modified. It should contain code that copies the code from RAM to ROM. The application's entry point is located in the ROM library, so be sure to call it explicitly.

Application PRM File

The S-Record File (generated previously) must be linked to the application with an offset.

Suppose the application code must be placed at address 0x800 in ROM and should be copied to address 0x7000 in RAM. The application's PRM file looks as in Listing A.23.

Listing A.23 Linker Parameter File

```
LINK fiborom.abs

NAMES  mystart.o fiboram.abs ansis.lib END
SECTIONS

    MY_RAM = READ_WRITE 0x5000 TO 0x53FF;
    MY_ROM = READ_ONLY  0x0600 TO 0x07FF;
PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS  INTO MY_ROM;
    DEFAULT_RAM                    INTO MY_RAM;
END
STACKSIZE 0x100
VECTOR 0 _Startup /* set reset vector on startup function */
HEXFILE fiboram.s1 OFFSET 0xFFFF9800 /* 0x800 - 0x7000 */
```

NOTE The offset specified in the HEXFILE command is added to each record in the S-Record File. The code at address 0x700 is encoded at address 0x800.

If CodeWarrior is used, then the CodeWarrior IDE will pass all the names in the NAMES...END directive directly to the linker. Therefore, the NAMES...END directive should be empty.

Copying Code from ROM to RAM

You must implement a function that copies the code from ROM to RAM.

Suppose the application code must be placed at address 0x800 in ROM and should be copied to address 0x7000 in RAM. You can implement a copy function that does this as in Listing A.24.

Listing A.24 Definition of the CopyCode() Function

```
/* Start address of the application code in ROM. */
#define CODE_SRC  0x800

/* Destination address of the application code in RAM. */
#define CODE_DEST 0x7000
```

```
#define CODE_SIZE 0x90 /* Size of the code which must be copied.*/

void CopyCode(void) {
  unsigned char *ptrSrc, *ptrDest;

  ptrSrc = (unsigned char *)CODE_SRC;
  ptrDest = (unsigned char *)CODE_DEST;
  memcpy (ptrDest, ptrSrc, CODE_SIZE);
}
```

Invoking the Application's Entry Point in the Startup Function

The startup code should call the application's entry point, which is located in the ROM library. You must explicitly call this function by its name. The best place is just before calling the application's main routine (Listing A.25).

Listing A.25 Invoking the Application's Entry Point

```
void _Startup(void) {
  ... set up stack pointer ...
  ... zero out ...
  ... copy down ...
  CopyCode();
  ... call main ...
}
```

Defining a dummy main function

The linker cannot link an application if there is no main function available. As in our case, the ROM library contains the main function. Define a dummy main function in the startup module (Listing A.26).

Listing A.26 Definition of a dummy main Function

```
#pragma NO_ENTRY
#pragma NO_EXIT
void main(void) {
  asm NOP;
}
```

Frequently Asked Questions (FAQs), Troubleshooting

This section provides some tips on how to solve the most commonly encountered problems.

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:

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.

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.

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 Listing A.27:

Listing A.27 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 Listing A.28 is FALSE:

Listing A.28 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.

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.

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.

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.

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.

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`, `C:\win95` or `C:\winnt`) or in its system (Win32s) or system32 (Windows 95, 98, Me, NT, 2000, XP, or 2003) subdirectory. The Win32s distribution also contains `Winhlp32.exe`.

To change the association with Windows 95 or Windows NT either (1) use the explorer menu "*View->Options*" and then the "*`File Types`*" tab or (2) select any help file and press the *Shift* key. Hold it while opening the context menu by clicking on the right mouse button. Select "*Open with ...*" from the menu. Enable the "*Always using this program*" 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.

How can constant objects be allocated in ROM?

Use #pragma INTO_ROM: Put Next Variable Definition into ROM and the -Cc: Allocate Constant Objects into ROM compiler option.

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. In addition, you can use the -I: Include File Path compiler option to specify the include file path. With CodeWarrior, check the access path in the preference panel.

How can I switch off smart linking?

By adding a '+' after the object in the NAMES list of the prm file.

With CodeWarrior and the ELF/DWARF object-file format (see -F (-Fh, -F1, -F1o, -F2, -F2o,-F6, or -F7): Object-File Format) compiler option, you can link all in the object within an `ENTRIES...END` directive in the linker prm file:

```
ENTRIES fibo.o:* END
```

This is NOT supported in the HIWARE object-file format.

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’.

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’.

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
```

How can an overview of all the compiler options be produced?

Type in -H: Short Help on the command line of the compiler.

How can a custom startup function be called after reset?

In the prm file, use:

```
INIT myStartup
```

How can a custom name for the main() function be used?

In the prm file, use:

```
MAIN myMain
```

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
```

How can the compiler be configured for the editor?

Open the compiler, select *File* > *Configuration* from the menubar, and choose Editor Settings.

Where are configuration settings saved?

In the `project.ini` file. With CodeWarrior, the compiler settings are stored in the `*.mcp` file.

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.

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.

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 Constant Objects into ROM option.

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.

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.

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.

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.

I cannot connect to my target board using an ICD Target Interface.

Communication may fail for the following reasons:

- Is the parallel port working correctly? Try to print a document using the parallel port. This allows you to ensure that the parallel port is available and connected.
- Is the BDM connector designed according to the specification from P&E?
- If you are running a Windows NT or Win98 operating system, you need to install an additional driver in order to be able to communicate with the software. See section NT Installation Notice in the debugger ICD Target Interface Manual.

- The original ICD Cable from P&E should not be extended. Extending this cable can often generate communication problems. The cable should not be longer that the original 25 cm.
- Maybe the PC is too fast for the ICD cable. You can slow down the communication between the PC and the Target using the environment variable BMDELAY (e.g., BMDELAY=50).

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 A.29 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

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**.

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`.

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."

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...)

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.

Production End

A production is always terminated by a period.

EBNF Syntax

The definition of EBNF in the EBNF language is:

Listing A.30

```
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.

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 Listing A.31.

Listing A.31 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 Listing A.31, where any character can be inserted).

Abbreviations, Lexical Conventions

Table A.1 has some programming terms used in this manual.

Table A.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

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 A.2):

Table A.2 Supported number suffixes

Constant	Suffix	Type
floating	`F`	`float`
floating	`L`	`long double`
integral	`U`	`unsigned in t`
integral	`uL`	`unsigned long`

Suffixes are not case-sensitive, e.g., ‘`ul`’, ‘`Ul`’, ‘`uL`’ and ‘`UL`’ all denote an `unsigned long` type. Listing A.32 has examples of these numerical formats.

Listing A.32 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 */
```

Precedence and Associativity of Operators for ANSI-C

Table A.3 gives an overview of the precedence and associativity of operators.

Table A.3 ANSI-C Precedence and Associativity of Operators

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right

Table A.3 ANSI-C Precedence and Associativity of Operators (*continued*)

Operators	Associativity
`? :`	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 A.33 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 Listing A.33, operator-precedence causes the product of (c*d) to be added to b, and that sum is then assigned to a.

In Listing A.34, 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 A.34 3 assignments in 1 statement

```
a = b += c += 1;
```

List of all Escape Sequences

Table A.4 gives an overview over escape sequences which could be used inside strings (e.g., for printf):

Table A.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`

B

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

[Options] Section

This section documents the entries that can appear in the `[Options]` section of the file `mcutools.ini`.

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
```

[*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 HC12 compiler, the name of this section would be [HC12_Compiler].

SaveOnExit

Arguments

1/0

Description

Set to 1 if the configuration should be stored when the compiler is closed. Set to 0 if it should not be stored. The compiler does not ask to store a configuration in either case.

SaveAppearance

Arguments

1/0

Description

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The command line, its history, the windows position, and other topics belong to this entry.

SaveEditor

Arguments

1/0

Description

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The editor setting contains all information of the Editor Configuration dialog box.

SaveOptions

Arguments

1/0

Description

Set to 1 if the options should be saved when writing a project file. Set to 0 if the options should not be saved. The options also contain the message settings.

RecentProject0, RecentProject1, ...

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
```

TipFilePos

Arguments

Any integer, e.g., 236

Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

Saved

Always saved when saving a configuration file.

ShowTipOfDay

Arguments

0/1

Description

Should the Tip of the Day dialog box be shown at startup.
1: It should be shown
0: Only when opened in the help menu

Saved

Always saved when saving a configuration file.

TipTimeStamp

Arguments

date and time

Description

Date and time when the tips were last used.

Saved

Always saved when saving a configuration file.

[Editor] Section

This section documents the entries that can appear in the `[Editor]` section of the `mcutools.ini` file.

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.

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.

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
```

Example

Listing B.1 shows a typical `mcutools.ini` file.

Listing B.1 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
```

C

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 `project.ini` file can contain these sections:

- [Editor] Section
- [XXX_Compiler] Section
- Example

[Editor] Section

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.

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.

Editor_Opts

Arguments

Local editor options

Description

Specifies options that should be used for the local 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. This entry has the same format as the global Editor Configuration in the `mcutools.ini` file.

Example [Editor] Section

```
[Editor]
editor_name=notepad
```

```
editor_exe=C:\windows\notepad.exe
editor_opts=%f
```

[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 HC12 compiler, the name of this section would be [HC12_Compiler].

RecentCommandLine*X*

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.

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.

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.

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.

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.

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
```

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.

EditorType

Arguments

0/1/2/3

Description

This entry specifies which Editor Configuration is active.
0: Global Editor Configuration (in the file `mcutools.ini`)
1: Local Editor Configuration (the one in this file)
2: Command line Editor Configuration, entry EditorCommandLine
3: DDE Editor Configuration, entries beginning with EditorDDE

For details see Editor Configuration.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

EditorCommandLine

Arguments

Command line for the editor.

Description

Command line content to open a file. For details see Editor Configuration.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog box.

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.

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.

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.

Example

Listing C.1 shows a typical configuration file layout (usually `project.ini`):

Listing C.1 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
```

D

Using the Linux Command Line Compiler

This appendix documents the HC12 Compiler command line program. The compiler program is named `chc12` and is located in the `prog` subfolder of the CodeWarrior installation path. The compiler program can be ran from a shell command line or specified in a makefile.

Command Line Arguments

Enter `chc12 -h` to display a list of available arguments and options. Compiler options are described in the chapter “Compiler Options”. The color setting options such as `WmsgCE` are available for the Windows operating system only.

Command Examples

The following examples demonstrate some simple uses of the linux version of the HC12 command line compiler.

One method of setting paths to library files is to use the `-Env` option with the variable `LIBPATH` with a colon-separated list of directories.

```
chc12 main.c -Env”LIBPATH=/usr/lib;/usr/bin/lib”
```

To set the maximum number of error messages to 5 and create the `err.log` error file and a listing file in the current directory enter:

```
chc12 main.c -WmsgNe5 -WErrFileOn -Lasm
```

Using a Makefile

The `maker` command allows you to control and define the build process. The `maker` program reads a file called makefile or Makefile. This file determines the relationships between the source, object and executable files.

Once you have created your Makefile and your corresponding source files, you are ready to use the `maker` command. If you have named your Makefile either Makefile or makefile, `maker` will recognize it. If `maker` does not recognize your makefile or it uses a different name, you can specify `maker -f mymakefile`. The order in which dependencies are listed is important. If you simply type `maker` and then return, `maker` will attempt to create or update the first dependency listed.

The makefile has instructions for a specific project. Following is a sample makefile for an example application called banked_data used with the `maker` command and an explanation of some of the assignments.

```
#----------------------------------------------------------------
# HCS12X example
#----------------------------------------------------------------


#----------------------------------------------------------------
# Change the following paths with the appropriate paths for your
# machine.
#----------------------------------------------------------------
TOOLS_PATH=/home/sources/X/prog
HC12_LIB=/home/sources/X/lib


APP_NAME=banked_data.abs
PRMFILE=prm/Simulator_linker.prm
BBLFILE=prm/burner.bbl
BUILDLOG=build.log

#----------------------------------------------------------------
# Tools definition
#----------------------------------------------------------------
CC   = $(TOOLS_PATH)/chc12
LD   = $(TOOLS_PATH)/linker
BURN = $(TOOLS_PATH)/burner
#----------------------------------------------------------------
# Build tool options
#----------------------------------------------------------------
CFLAGS = -I"$(HC12_LIB)/hc12c/include"  -CPUHCS12X -D__NO_FLOAT__
-D__FAR_DATA -Mb -PSegObj
LD_FLAGS  = -M
#----------------------------------------------------------------
CFILES = mc9s12xdp512.c main.c datapage.c start12.c bankeddata.c
#----------------------------------------------------------------


#----------------------------------------------------------------
BINDIR=bin
```

```
OFILES = $(patsubst %.c,$(BINDIR)/%.o,$(filter %.c,$(CFILES)))
OFILES += $(patsubst %.cpp,$(BINDIR)/%.o,$(filter %.cpp,$(CFILES)))
VPATH = $(PWD)/src
#-----------------------------------------------------------------

#-----------------------------------------------------------------
# Required libraries
#-----------------------------------------------------------------
LIBS = "$(HC12_LIB)/hc12c/lib/ansixbi.lib"
#-----------------------------------------------------------------


#-----------------------------------------------------------------
# Targets
#-----------------------------------------------------------------
absfile: .INIT $(OFILES)
       @echo -n "Linking ..."
       @$(LD) $(PRMFILE) $(COMMON_FLAGS) $(LD_FLAGS) -Add{$(LIBS)}
         -Add{$(OFILES)} -O${APP_NAME} >> $(BUILDLOG)
       @echo "done"

srec: absfile
    @echo -n "Generating srecord ..."
    @$(BURN) -Env"ABS_FILE=$(APP_NAME)" -f $(BBLFILE) >> $(BUILDLOG)
    @echo "done"
$(BINDIR)/%.o : %.c
    @echo -n "*** Compiling $< ... -->$@ ..."
    @$(CC) $(CFLAGS) -objn="$@" $< >> $(BUILDLOG)
    @echo "done"

$(BINDIR)/mc9s12xdp512.o:$(HC12_LIB)/hc12c/src/mc9s12xdp512.c
    @echo -n "*** Compiling $< ... -->$@ ..."
    @$(CC) $(CFLAGS) -objn="$@" $< >> $(BUILDLOG)
    @echo "done"

.INIT :
    @if [ ! -e $(BINDIR) ];then mkdir $(BINDIR);fi
    @if [ -e $(BUILDLOG) ];then rm -f $(BUILDLOG);fi
    @echo $(OFILES)

#-----------------------------------------------------------------
# Cleanup
#-----------------------------------------------------------------
clean:
       -rm -f $(OFILES)
       -rm -f *.abs
       -rm -f *.map
       -rm -f *.bpt
       -rm -f *.mrk
       -rm -f *.log
       -rm -f *.phy
       -rm -f *.s19
       -rm -f *.map
```

You will notice in the makefile that the HC12 compiler and linker programs are assigned to the CC and LD macros under the tools definition commented section.

```
#------------------------------------------------------
# Tools definition
#------------------------------------------------------
CC   = $(TOOLS_PATH)/chc12
LD   = $(TOOLS_PATH)/linker
```

The final binary executable file is also specified in the makefile.

```
APP_NAME=banked_data.abs
```

You can examine compiler errors and warnings in a specified log file, for example, `build.log` is specified in the makefile. Common errors occur when include files or source files cannot be found. Make sure that path assignments are correct and accessible.

Using the .hidefaults File

A `.hidefaults` file can be used to set environment variables. A sample file looks as follows:

```
OBJPATH=./bin
TEXTPAH=./bin
GENPATH=/home/sources/X/lib/hc12c/include;/home/sources/X/lib/hc12c
/src;./src;./prm
LIBPATH=/home/sources/X/lib/hc12c/include
```

Index