C++ *Version 1.0*

For the ARM Software Development Toolkit

User and Reference Guide

Document Number: ARM DUI 0047 A
December 1997

ENGLAND
Advanced RISC Machines Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

ARM
Advanced RISC Machines

Proprietary Notice

User and Reference Guide

ARM DUI 0047 A

Preface

This preface introduces ARM C++ version 1.0 and its documentation. It contains the following sections:

About This Book

Overview

This book describes release 1.0 of ARM C++ for the ARM Software Development Toolkit, hereafter referred to as ARM C++. ARM C++ is a suite of tools that, when used in conjunction with the ARM Software Development Toolkit version 2.11 or later, allow you to develop C++ applications for the ARM family of RISC processors.

This book covers information specific to ARM C++ only. It is intended to be used in conjunction with the ARM Software Development Toolkit version 2.11 documentation suite.

Organization

This book is organized into the following chapters:

Chapter 1 **Introduction**

Read this chapter for an introduction to ARM C++.

Chapter 2 **Using the ARM Project Manager with C++**

Read this chapter to learn about the additions that ARM C++ makes to the SDT 2.11 ARM Project Manager.

Chapter 3 **Using the ARM Debugger for Windows with C++**

Read this chapter to learn about the additions that ARM C++ makes to the SDT 2.11 ARM Debugger for Windows.

Chapter 4 **Using the ARM C++ Compilers**

Read this chapter for information on how to use the ARM C++ compilers. This chapter includes descriptions of the compiler options, and other compiler-specific features.

Chapter 5 **Mixed Language Programming**

Read this chapter for a brief introduction to writing mixed C, C++, and Assembler code. This chapter includes a description of using inline Assembler from C++.

Chapter 6 **ARM C++ Compiler Reference**

Read this chapter for reference information about the ARM C++ compiler and how it implements the C and C++ languages. This chapter includes information on compiler limits, and language conformance.

Related Publications

This book contains information that is specific to ARM C++. For additional information on the ARM Software Development Toolkit, please refer to the following ARM publications:

- ARM Software Development Toolkit Version 2.11 *User Guide* (ARM DUI 0040C)
- ARM Software Development Toolkit Version 2.11 *Reference Guide* (ARM DUI 0041B)
- The ARM *Architectural Reference Manual* (ARM DDI 0100B)

Further reading

This book is not intended to be an introduction to C++ and does not try to teach programming in C++, nor is it a reference manual for the C++ standard. The following texts provide general information on C++.

ISO/IEC C++ reference

- ISO/IEC JTC1/SC22 *Final CD (FCD) Ballot for CD 14882: Information Technology - Programming languages, their environments and system software interfaces - Programming Language C++*

  This is the December 1996 version of the draft ISO/IEC standard for C++. It is referred to hereafter as the *Draft Standard.*

C++ programming guides

The following books provide general C++ programming information.

- Ellis, M.A. and Stroustrup, B., *The Annotated C++ Reference Manual* (1990). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-51459-1.

  This is a reference guide to C++.

- Stroustrup, B., *The Design and Evolution of C++* (1994). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-54330-3.

  This book explains how C++ evolved from its first design to the language in use today.

- Meyers, S., *Effective C++* (1992). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-56364-9.

  This provides short, specific, guidelines for effective C++ development.

- Meyers, S., *More Effective C++* (1996). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-63371-X.

  The sequel to *Effective C++*.

C programming guides

Because the ARM C++ compiler is also a compiler for ANSI C, the following books are relevant:

- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

  This is the original C bible, updated to cover the essentials of ANSI C.

- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.

  This is a very thorough reference guide to C, including useful information on ANSI C.

- Koenig, A, *C Traps and Pitfalls,* Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

  This explains how to avoid the most common traps and pitfalls in C programming. It provides informative reading at all levels of competence in C.

ANSI C reference

- ISO/IEC 9899:1990, *C Standard*

  This is available from ANSI as X3J11/90-013. The standard is available from the national standards body (for example, AFNOR in France, ANSI in the USA).

Typographical conventions

The following typographical conventions are used in this book:

`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.
`type`writer	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands and functions.
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.
small bold	Denotes language keywords when used outside example code.

Feedback

Feedback on this book

If you have any comments about this document, please contact your supplier giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Feedback on ARM C++ for the Software Development Toolkit

If you have any problems with ARM C++, please contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small stand-alone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Contents

Chapter 1

Introduction

This chapter introduces ARM C++ version 1.0 for the ARM Software Development Toolkit. It contains the following sections:

1.1 About ARM C++

ARM C++ consists of a set of tools and enhancements to the ARM Software Development Toolkit (SDT). To use ARM C++ you must have version 2.11 or later of the SDT. ARM C++ enables you to develop applications for the ARM family of processors in C, C++, and ARM assembler.

ARM C++ provides two variant compilers:

armcpp compiles C, C++, and Assembler code into 32 bit ARM code

tcpp compiles C, C++, and Assembler code into 16-bit Thumb code.

Refer to Chapter 4 for detailed information on using the ARM C++ compilers. Refer to Chapter 5 for information on writing mixed C, C++, and ARM assembler code.

ARM C++ provides support for most of the ISO/IEC JTC1/SC22 December 1996 Draft Standard for C++. This is referred to throughout this book as the *Draft Standard*.

ARM C++ provides Standard C++ library support in the form of the *Rogue Wave Standard C++ library* version 1.2.1, together with the ARM Standard C library and additional C++ library functions. Refer to Chapter 6 for more information on ARM C++ library and language support.

Additions to the ARM Project Manager (APM) provide support for building C++ projects in APM. These additions provide C++ project templates for building ARM, Thumb, and ARM/Thumb interworking projects. Refer to Chapter 2 for more information on APM.

Additions to the ARM Debugger for Windows (ADW) provide support for debugging your C++ code using ARM Graphical User Interface tools. ADW now provides:

- support for the DWARF 2 debug table format
- a C++ interpretation of class hierarchies, structures, and variables.

Refer to Chapter 3 for information on ADW.

1.2 Release Components

This section describes the major release components of ARM C++.

1.2.1 Tools

ARM C++ provides the following tools:

armcpp the ARM 32-bit compiler

tcpp the Thumb 16-bit compiler

adw.exe this is a replacement for the SDT 2.11 `adw.exe`

adw_cpp.dll provides additional functionality to ADW to support debugging

armsd this is a replacement for the SDT 2.11 `armsd`.

1.2.2 Libraries

ARM C++ includes the precompiled ARM C++ library. This consists of:

Rogue Wave Standard C++ library version 1.2.1

This is provided as prebuilt binaries. Refer to the release notes for information on obtaining the source code for this library. The Rogue Wave Standard C++ library documentation is supplied in HTML format.

ARM C++ library functions

Additional ARM library support functions. These are provided in both source and binary format.

1.2.3 Other components

APM templates A set of templates to support building C++ projects in APM.

1.3 Supported Platforms

ARM C++ is supported on the following platforms.

1.3.1 Command-line development tools

The command-line development tools are supported on:

- Sun Workstations running SunOS 4.1.3 or Solaris 2.4 or 2.5
- Hewlett Packard workstations running HP-UX 9.0
- IBM compatible PCs running Windows 95
- IBM compatible PCs running Windows NT 3.51 or 4.0
- Digital Alpha PCs running Windows NT 3.51 or 4.0.

1.3.2 Windows development tools

The Windows development tools are supported on:

- IBM compatible PCs running Windows 95
- IBM compatible PCs Windows NT 3.51 or 4.0.

Chapter 2

Using the ARM Project Manager with C++

This chapter describes how to use the ARM Project Manager (APM) with ARM C++. It also describes the APM templates distributed with Release 1.0 of ARM C++. It contains the following sections:

2.1 About the APM for C++

ARM C++ provides project templates to enable you to build C++ projects in APM. The C++ project templates are based on the corresponding C project templates for the Software Development Toolkit version 2.11. The templates provide options for producing C++ executable images and object libraries from within APM.

By default, the templates are installed in the `\template` directory of your SDT 2.11 installation directory. If you installed SDT 2.11 in the default location, this will be `C:\ARM211\Template`.

The C++ APM templates are:

ARM C++ Executable Image

This template builds an ARM C++ executable image from C, C++, and ARM Assembler source files, and ARM Object libraries. The template is installed as `A01AEICPP.apj`

Thumb C++ Executable Image

This template builds a Thumb C++ executable image from C, C++, and Thumb/ ARM Assembler source files, and Thumb object libraries. The template is installed as `A02TEICPP.apj`

ARM C++ Object Library

This template builds an ARM object library file from C, C++, and ARM Assembler source files. You can use the library as a component in other projects to build ARM executable images. The template is installed as `A03AOLCPP.apj`

Thumb C++ Object Library

This template builds a Thumb object library file from C, C++, and Thumb/ARM Assembler source files. You can use the library as a component in other projects to build Thumb executable images. The template is installed as `A04TOLCPP.apj`

Thumb/ARM C++ Interworking Image

This template builds an ARM/Thumb C++ interworking image from:

- Thumb C and C++ source files
- ARM C and C++ source files
- Thumb/ARM Assembler source files
- Thumb Object Libraries
- ARM Object Libraries.

The template is installed as `A05TAIICPP.apj`

In all functional respects, the C++ version of APM is identical to the SDT 2.11 release. Please refer to Chapter 2 of the SDT 2.11 *User Guide* for more information on using those parts of APM that are not specific to C++.

2.2 Using the ARM Project Manager C++ Templates

The APM C++ templates provide a number of new options for creating C++ source files, `#include` files, and projects. The following sections describe these options.

2.2.1 Creating new projects and source files

This section describes how to create new projects and source files based on the C++ project templates. The following general points apply to the templates:

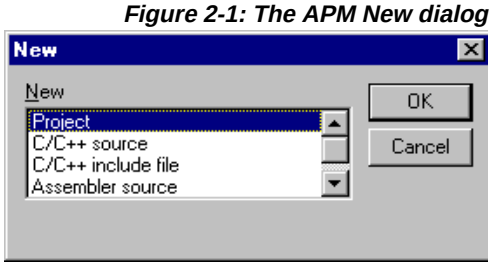
- All templates that produce ARM executable images or ARM object libraries are configured to use armcpp to compile C++ source files.
- All templates that produce Thumb executable images or Thumb object libraries are configured to use tcpp to compile C++ source files.
- All templates that produce executable images use the ARM Debugger for Windows (ADW) as their debugger.
- You can convert a non-interworking project to an interworking project by following the instructions in section 12.7 of the SDT 2.11 *User Guide.* Substitute `armcpp` where the *User Guide* refers to `armcc`, and `tcpp` where the *User Guide* refers to `tcc`.
- Libraries must contain either ARM code only, or Thumb code only.

Creating a new project

Follow these steps to create a new C++ project:

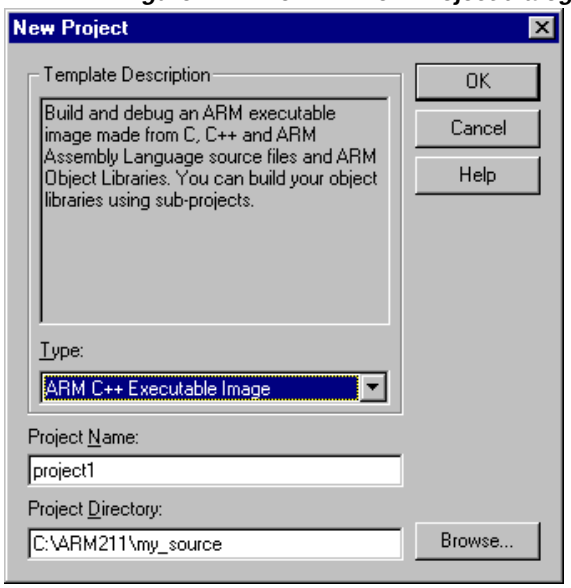
1. Select **New...** from the **File** menu. The New dialog is displayed (Figure 2-1).

Figure 2-1: The APM New dialog



2. Select **Project** from the list of options and click **OK**. The New Project dialog is displayed (Figure 2-2).

Figure 2-2: The APM New Project dialog



3. Select the type of project you wish to create. In addition to the standard options available in SDT 2.11, you can create a project based on the new C++ templates. These are:

   - ARM C++ Executable Image
   - Thumb C++ Executable Image
   - ARM C++ Object Library
   - Thumb C++ Object Library
   - Thumb/ARM C++ Interworking Image.

4. Enter a project name and project directory for the new project.

5. Click **OK**. A new project is created for the type of image or library you have chosen.

Creating new source and include files

Follow these steps to create a new source file or include file:

1. Select **New...** from the **File** menu. The New dialog is displayed (Figure 2-1).

2. Select the type of source file you wish to create from the list of options and click **OK**. A new file of the selected type is opened.

2.3 Configuring a C++ Project

This section describes APM configuration options. Note that this section covers only those options that are new with the ARM C++ version of APM. Refer to section 2.5 of the SDT 2.11 *User Guide* for information on configuration options that are unchanged from the SDT 2.11 version of APM.

Changes to project configuration can be either system-wide or project-specific. For this release of ARM C++, the compiler is the only tool that has C++ specific configuration options. Other tools, such as the ARM assembler and the linker, are identical to their SDT 2.11 counterparts.

2.3.1 Configuring the compiler

Compiler options are set in the Compiler Configuration window. You can set compiler options for:

- specific projects
- all projects (system-wide)
- specific build variants
- individual source files.

Setting project-specific options

To set project-specific options:

1. Open the project window for the project you wish to configure.
2. Select the level of the project hierarchy you wish to configure. For example:
   - click on the top level project icon to set options for the whole project
   - click on the icon for a build variant, such as the Debug variant, to set options for that specific build variant
   - click on the icon for a project source file to set options for that specific source file.
3. Select **Tool configuration for *element_name*** from the **Project** menu, where *element_name* is the name of the project, build variant, or source file you have selected in step 2.
4. Select **Set** from the **<ccpp> = *toolname*** sub-menu where *toolname* is either armcpp or tcpp, depending on the project template you are using.

   The Compiler Configuration window is displayed. Figure 2-3 on page 2-7 shows an example. Any changes you make in this window will affect only the project, build variant, or source file you have specified.
5. Click on the tab for the compiler option you wish to change. Tabs that contain C++ specific options are:
   - Target
   - Warnings
   - Errors
   - C++ and Debug.

   These options are described in more detail in the sections below.

Setting system-wide options

To set system-wide options:

1. Select **Configure** from the **Tools** menu.
2. Select **<ccpp> = *toolname***, where *toolname* is either armcpp or tcpp, depending on the project template you are using. A warning message is displayed asking you to confirm that you wish to change the global configuration options for the tool.
3. Click **Yes** to continue. The Compiler Configuration window is displayed. Figure 2-3 on page 2-7 shows an example. Any changes you make in this window will affect all C++ projects that use the compiler you are configuring.

Note *If you move a project from one SDT installation to another, it may not behave as expected unless the same system-wide options are set for each.*

4. Click on the tab for the compiler option you wish to change. Tabs that contain C++ specific options are:
   - **Target**
   - **Warnings**
   - **Errors**
   - **C++ and Debug**.

   These options are described in more detail in the sections below.

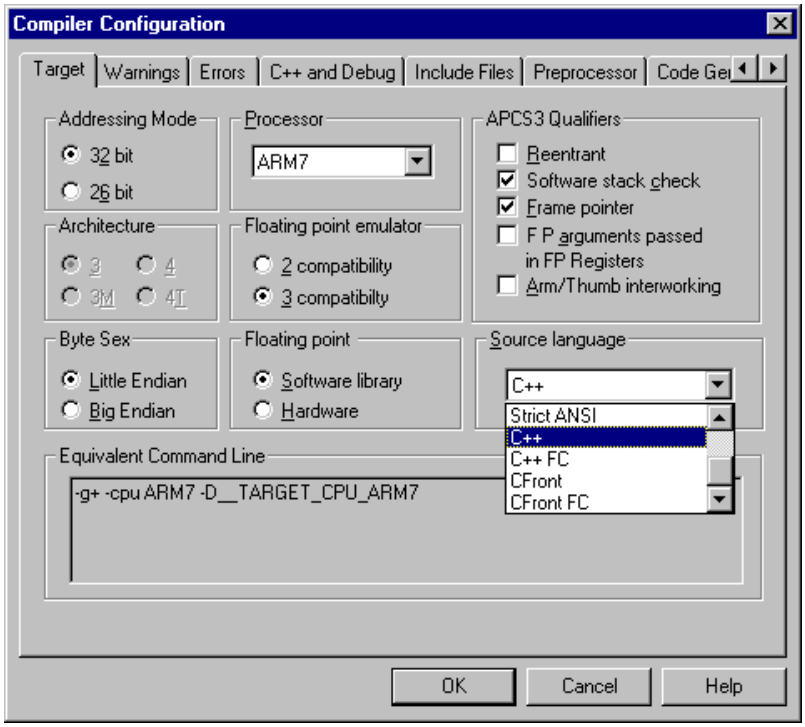
Note *If you want to use the C compiler for a particular file or partition, click on the file or partition and change the value of the ccpp variable from armcpp to armcc.*

Configuring the target

Follow these steps to configure C++ specific options for generating code for a specific target processor or architecture:

1. Click on the **Target** tab in the Compiler Configuration window. The target options are displayed (Figure 2-3). Note that these options are identical to those in SDT 2.11 except that the **Source language** drop down list contains additional options for C++.

Figure 2-3: Configuring the target processor



2. Select the source language for the project. The C++ specific options are:

   C++ Select this option to compile draft-conforming C++ code.

   C++ FC Select this option to enable limited PCC mode. This mode allows you to use PCC-style header files in an otherwise strict ANSI C or C++ mode. It supports system programming, and allows you to use libraries of functions implemented in old-style C from an application written in draft-conforming C++.

   The limited PCC option:

   - Allows characters after `#else` and `#endif` preprocessor directives. The characters are ignored.
   - Suppresses warnings about explicit casts of integers to function pointers.
   - Permits the dollar character in identifiers. Dollar characters are normally reserved for use by the linker.

   CFront Select this option to alter the behavior of the compiler so that it is more likely to accept programs that Cfront accepts. Refer to *Setting the source language* on page 4-11 for a complete description of Cfront mode.

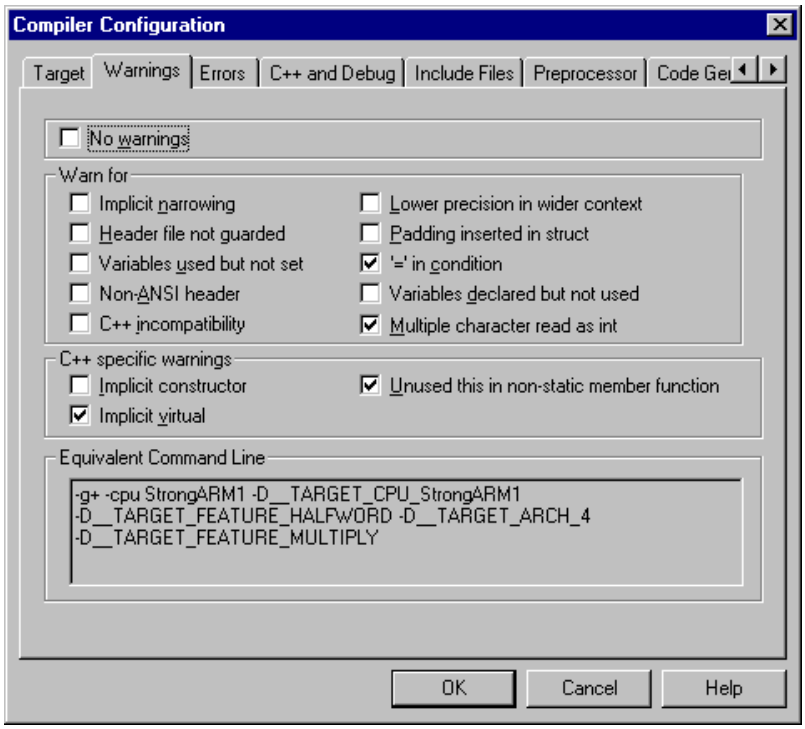
   CFront FC Select this option to combine Cfront and C++ FC modes. It allows Cfront-style code to use PCC-style header files.

Configuring warnings

Follow these steps to configure C++ specific options for warnings:

1. Click on the **Warnings** tab in the Compiler Configuration window. The options for warnings are displayed (Figure 2-4).

Figure 2-4: Configuring warnings



2. Select the warning options you require from the **C++ specific warning** section of the window. The options available are:

Implicit constructor

Select this option to suppress the implicit constructor warning. This warning is issued where a constructor would have to be invoked implicitly. For example:

```
struct X { X(int); };
X x = 10;                // actually means, X x = X(10);
                         // See the Annotated C
                         // Reference Manual p.272
```

Implicit virtual

Select this option to suppress the implicit virtual warning. This warning is issued when a non-virtual member function of a derived class hides a virtual member function of a parent class. For example:

```
struct Base { virtual void f(); };
struct Derived : Base { void f(); };
                         // warning 'implicit virtual'
```

Unused this in non-static member function

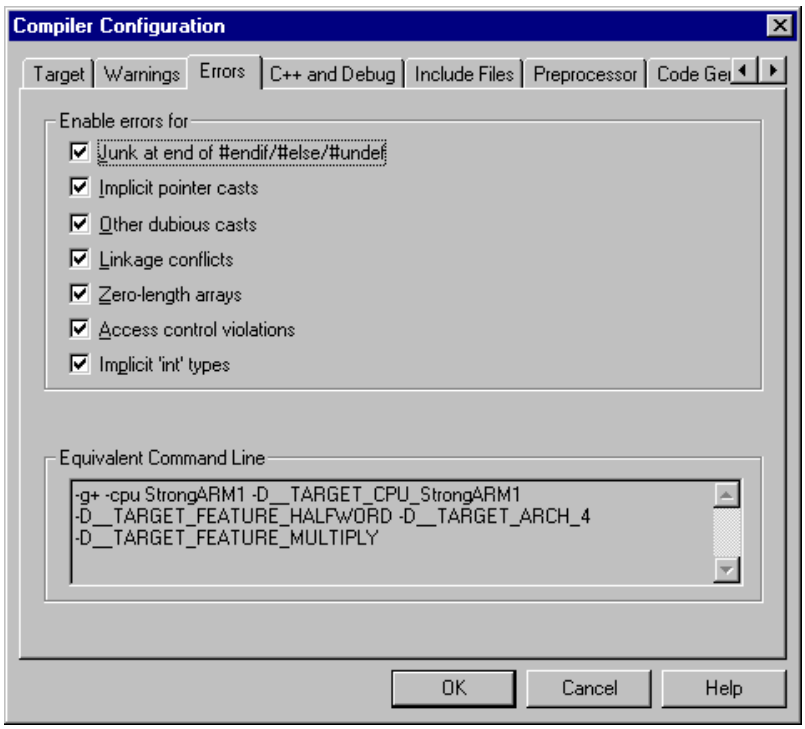
Select this option to suppress the unused ‘this’ warning. This warning is issued when the implicit 'this' argument is not used in a non-static member function.

Configuring errors

Follow these steps to configure C++ specific error options:

1. Click on the **Errors** tab in the Compiler Configuration window. The options for errors messages are displayed (Figure 2-5).

Figure 2-5: Configuring Errors



2. Select the error options you require. The following two options are specific to C++:

Access control violations

Select this option to convert access control errors to warnings. For example:

```
class A { void f() {}; };  // private member
A a;
void g() { a.f(); }        // erroneous access
```

Implicit ‘int’ types

Select this option to convert implicit ‘int’ assumption errors to warnings. For example:

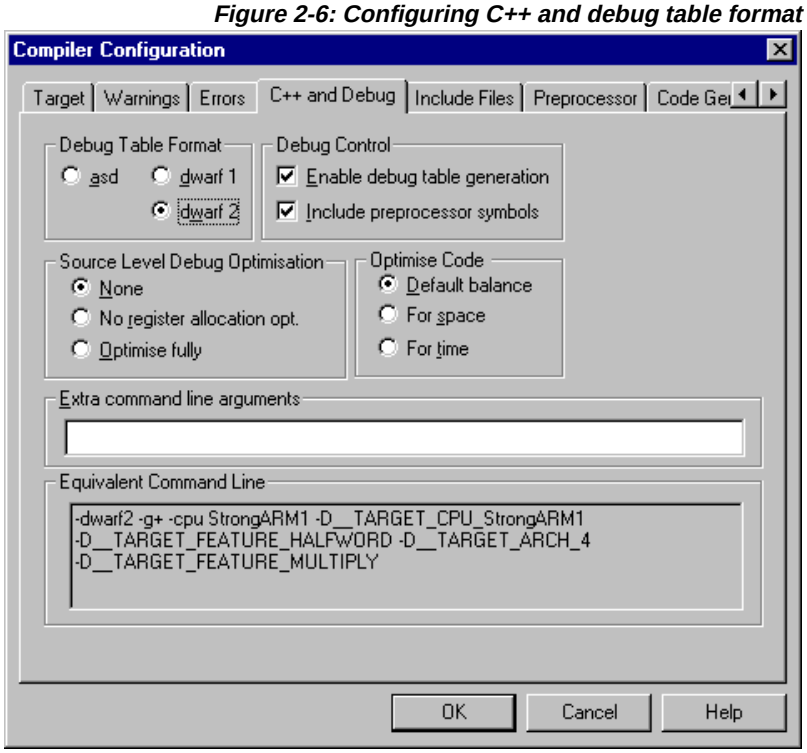
```
class c { const i; };
Error: declaration lacks type/storage-class
(assuming ’int’): ’i’
```

Configuring C++ and debug

Follow these steps to configure C++ specific options for the debug table format generated by the compiler:

1. Click on the **C++ and Debug** tab in the Compiler Configuration window. The options for debug and optimization are displayed (Figure 2-6).

Figure 2-6: Configuring C++ and debug table format



2. Select the debug and optimization options you require. For C++ projects the default is dwarf 2. Refer to *The debug table format* on page 3-11 for more information on setting debug table formats for C++.

Chapter 3

Using the ARM Debugger for Windows with C++

This chapter describes the additions that ARM C++ makes to the SDT 2.11 ARM Debugger for Windows (ADW). It does not describe those parts of ADW that are the same as the SDT 2.11 release. Please refer to the SDT 2.11 *User Guide* for more information on using those parts of ADW that are not specific to ARM C++. This chapter contains the following sections:

3.1 About ADW for C++

ARM C++ provides an enhanced version of ADW (`adw.exe`) to support C++ debugging. In addition, a dynamic link library (`adw_cpp.dll`) is installed in the same directory as `adw.exe`. By default, the installation directory is `c:\ARM211\bin`. The `adw_cpp.dll` adds:

- A **C++** menu between the **View** and **Execute** menus in the ADW main menu bar
- Five new buttons in the ADW toolbar:


Evaluate Expression


View Classes

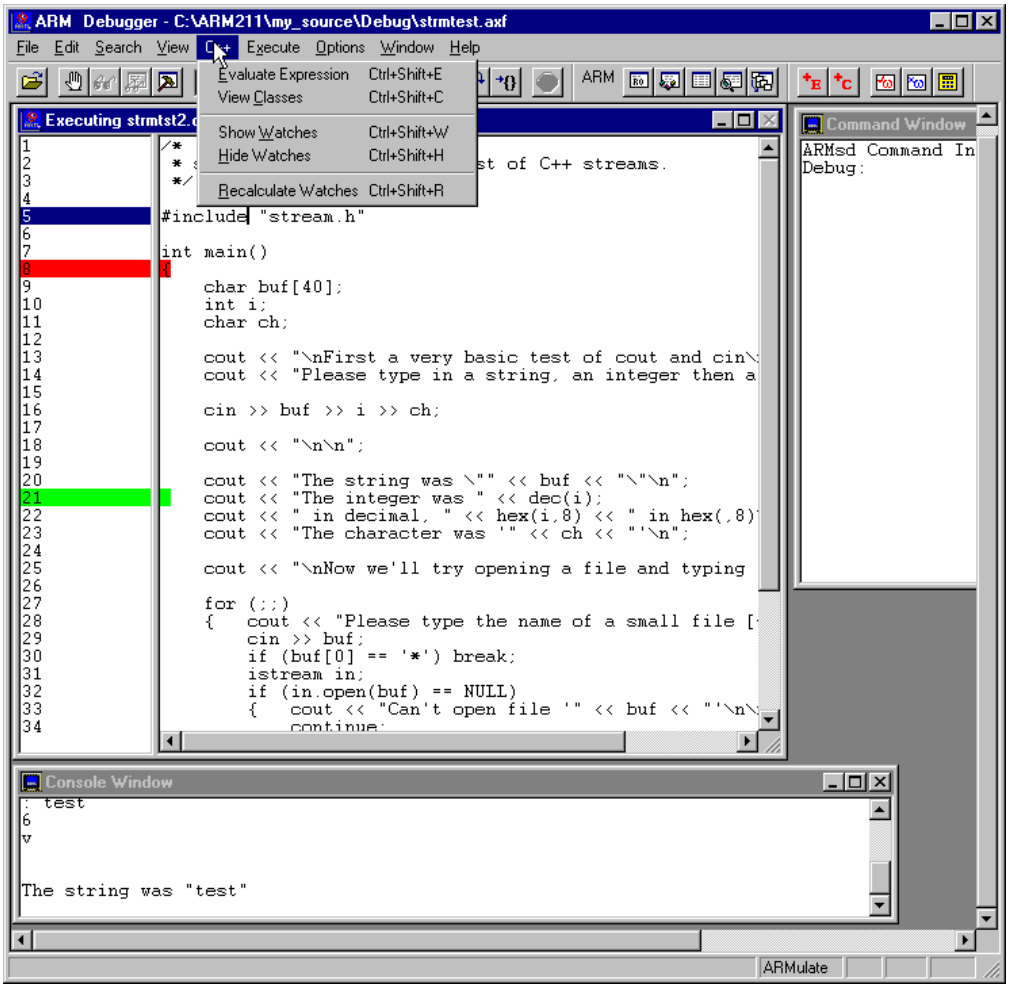

Show Watches


Hide Watches


Recalculate Watches

Figure 3-1 shows an example of the ADW C++ debug interface and the **C++** menu.

Figure 3-1: The ADW C++ interface



3.2 Using the C++ Debugging Tools

The menu items in the **C++** menu give access to three new ADW windows:

- The Class View window. This window displays the class hierarchy of a C++ program in outline format.
- The Watch View window. This window displays a list of watches. It allows you to add and remove variables and expressions to be watched, and change the contents of watched variables.
- The Evaluate Expression window. This window allows you to enter an expression to be evaluated, and to add that expression to the Watch window if you wish.

These windows are described in detail in the sections below.

3.2.1 Using the Class View window

The class view window enables you to view the class structure of your C++ program. Classes are displayed in an outline format that allows you to navigate through the hierarchy to display the member functions for each class. A special branch of the hierarchy called *Global* displays global functions.

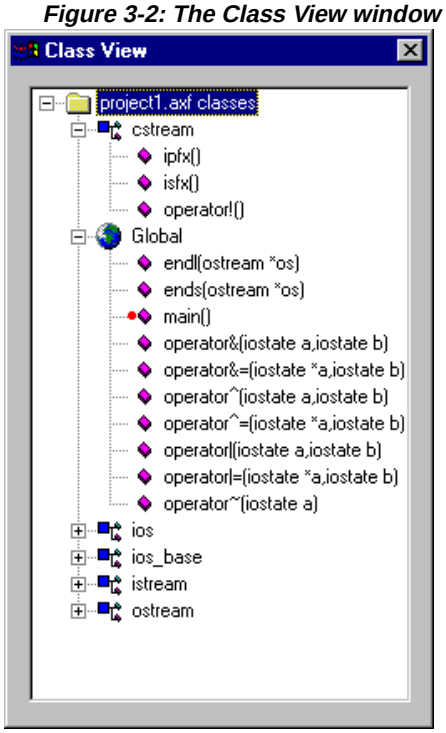
You can use the Class View window to view function code and set breakpoints for a class.

Displaying the Class View window

Follow these steps to open the Class View window:

1. Select **View Classes** from the **C++** menu, or click on the **View Classes** button in the toolbar. A Class View window is displayed that shows the class hierarchy of your C++ program. Figure 3-2 shows an example of the Class View window.

Figure 3-2: The Class View window

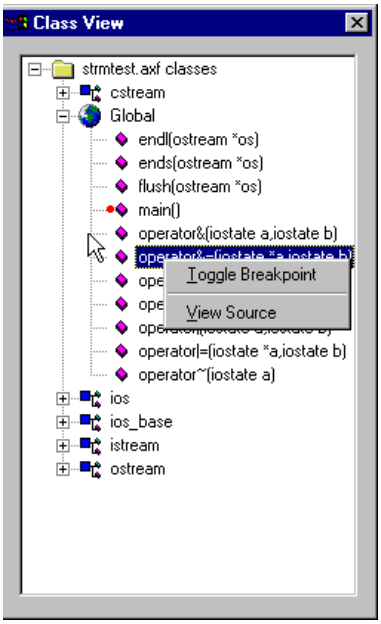


Viewing code from the Class View window

Follow these steps to view the source code for a class:

1. Display the Class View window.
2. Click the right mouse button on a member function. A **Class View** window menu is displayed (Figure 3-3).

Figure 3-3: The Class View window menu



3. Select **View Source** from the **Class View** window menu to display the source code for the function.

Note *You can also double click the left mouse button on a member function to display the function source.*

4. Select **Set or Edit Breakpoint...** from the **Execute** menu if you want to add a breakpoint within the code you are viewing. Refer to the next section for information on how to set a breakpoint at function entry.

Setting and clearing breakpoints from the Class View window

Follow these steps to toggle a breakpoint that will halt the program when the source for a class or function is entered:

1. Display the Class View window.
2. Click the right mouse button on a member function. A **Class View** window menu is displayed (Figure 3-3).
3. Select **Toggle Breakpoint** from the **Class View** window menu to set a breakpoint, or unset an existing breakpoint. Breakpoints are indicated by a red dot to the left of the function in the Class View window.

3.2.2 Using the Watch window

The Watch window allows you to set watches on variables and expressions. The Watch window provides similar functionality to the SDT 2.11 Local and Global windows. In addition, it provides a C++ interpretation of the data being displayed.

Note *The Watch window is not used to set watchpoints. Select **Set or Edit Watchpoint...** from the **Execute** menu to set watchpoints. Refer to the SDT 2.11 User Guide for more information.*

You can specify the contents and format of the Watch window using the **Watch** window menu. The following sections describe how to:

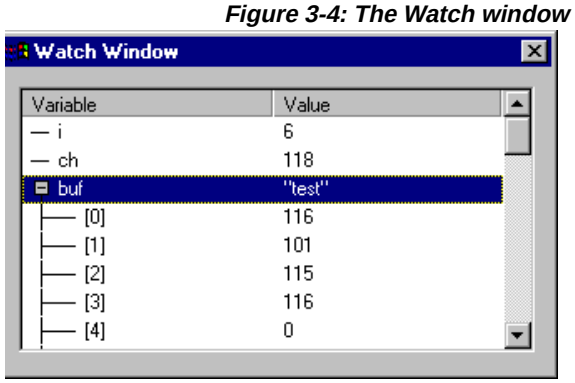
- view the Watch window
- display the **Watch** window menu
- delete and add watch items
- format watch items
- change the contents of watched items
- recalculate watches.

Viewing the Watch window

Follow these steps to view the Watch window:

1. Select **Show Watch Window** from the **C++** menu or click on the **Show Watches** button in the toolbar. The Watch window displays a list of watched variables and expressions. Figure 3-4 shows an example.

Figure 3-4: The Watch window



Expressions that return a scalar value are displayed as an expression-value pair. Non-scalar values, such as structures and classes, are displayed as a tree of member variables. If a class is derived, the base classes are represented by `::<base class>` member variables of the class.

Note *You can also open the Watch window from the Evaluate Expression window. Refer to Evaluating expressions on page 3-8 for more information.*

Displaying the Watch window menu

The **Watch** window menu enables you to add and delete watches, to change the display format of watches, and to change the contents of watched variables. Follow these steps to display the **Watch** window menu:

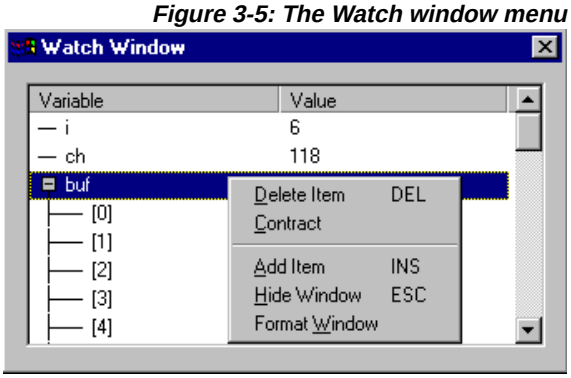
1. Display the Watch window.

2. Click the right mouse button in the Watch window. The **Watch** window menu is displayed. This menu is context sensitive. The menu items that it contains will depend on:

   - whether or not you have clicked on an existing watch item
   - the type of watch item you have clicked on.

   For example, Figure 3-5 shows the **Watch** window menu that is displayed when the right mouse button is clicked on the character array `buf`.

Figure 3-5: The Watch window menu



Deleting a watch item

Follow these steps to delete a watch item from the Watch window:

1. Display the Watch window.
2. Either:

   - click the right mouse button on the item you wish to delete and select **Delete Item** from the **Watch** window menu
   - click on the item you wish to delete and press the **DEL** key.

   The watch item is deleted from the Watch window.

Adding a watch item

Follow these steps to add a watch item to the Watch window:

1. Display the Watch window.
2. Either:

   - click the right mouse button in the Watch window to display the **Watch** window menu and select **Add Item** from the **Watch** window menu
   - press the **INS** key.

   A Watch Control window is displayed (Figure 3-6).

Figure 3-6: The Watch Control window

3. Enter an expression to add to the Watch window and click **OK**. Refer to *Evaluating expressions* on page 3-8 for more information on the types of expression you can add to the Watch window.

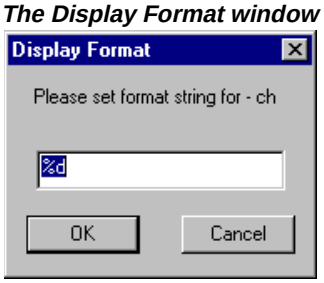
Note *You can also add an expression to the Watch window directly from the Evaluate Expression window. Refer to Evaluating expressions and adding watches on page 3-8 for more information.*

Formatting watch items

Follow these steps to change the formatting of values displayed in the Watch window:

1. Display the Watch window.
2. Click the right mouse button in the Watch window to display the **Watch** window menu.
3. Select **Format Item** from the **Watch** window menu to format a specific item. Alternatively, select **Format Window** to format all items in the window. The Display Format window is displayed (Figure 3-7).

Figure 3-7: The Display Format window



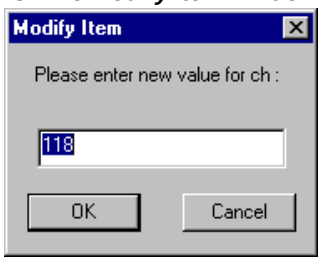
4. Enter a format string for the item, or items in the window. You can enter any single print conversion specifier that is acceptable as an argument to ANSI C `sprintf()` as a format string, except that * may not be used as a precision. For example, enter `%x` to format values in hexadecimal, or `%f` to format values as a character string.
5. Click **OK** to apply the format change.

Changing the contents of watched items

Follow these steps to change the contents of items in the Watch window:

1. Display the Watch window.
2. Click the right mouse button in the Watch window to display the **Watch** window menu.
3. Select **Edit value** from the **Watch** window menu. The Modify Item window is displayed (Figure 3-8).

Figure 3-8: The Modify Item window



4. Enter a new value for the variable.
5. Click **OK** to change the contents of the variable.

Recalculating watches

Select **Recalculate Watches** from the **C++** menu or click on the **Recalculate Watches** button in the toolbar to reinitialize the Watch window to its original state, with all structures and classes expanded by one level. This menu item can be used if the value of any variable may have been changed by external hardware while the debugger is not stepping through code.

3.2.3 Evaluating expressions

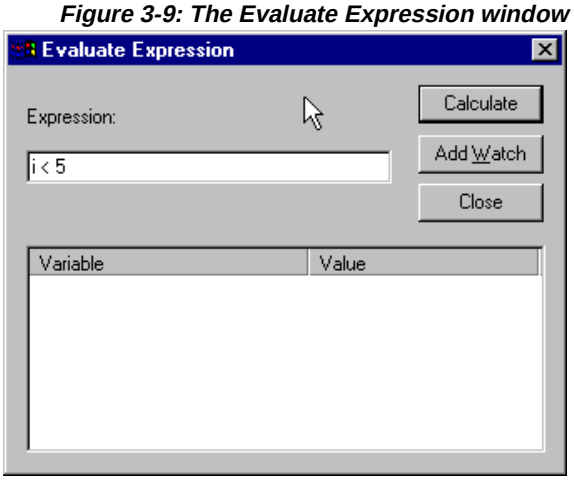
The Evaluate Expression window allows you to enter a simple C++ expression to be evaluated. The Evaluate Expression window provides similar functionality to the SDT 2.11 Expression window, with a C++ interpretation of the data being displayed.

Evaluating expressions and adding watches

Follow these steps to enter an expression to be evaluated:

1. Select **Evaluate Expressions** from the **C++** menu or click on the **Evaluate Expression** button in the toolbar. The Evaluate Expression window is displayed (Figure 3-9).

Figure 3-9: The Evaluate Expression window



2. Enter the expression to be evaluated and press the enter key, or click on the **Calculate** button. The value of the expression is displayed:
   - If the expression is a variable, the value of the variable is displayed.
   - If the expression is a logical expression, the window displays ‘1’ if the expression evaluates to true, or ‘0’ if the expression evaluates to false.

- If the expression is a function, the value of the function is displayed. Member functions of C++ classes cannot be evaluated.

Refer to *Expression evaluation guidelines* on page 3-9 for more information on expression evaluation in C++.

3. Click on the **Add Watch** button to add the expression to the Watch window.

Expression evaluation guidelines

Note *The following guidelines apply to all areas of ADW where an expression can be used, including setting watchpoints and breakpoints, and evaluating expressions in the Watch window.*

The following rules apply to expression evaluation for C++ :

- Member functions of C++ classes cannot be used in expressions.
- Overloaded functions cannot be used in expressions.
- Only C operators can be used in constructing expressions. For a full list of valid C operators, refer to section 10.5.3 of the SDT 2.11 *Reference Guide*. Any operators defined in a C++ class that also have a meaning in C (such as `[]`) will not work correctly because ADW uses the C operator instead. Specific C++ operators, such as the scope operator `::`, are not recognized.
- Member variables of a class cannot be accessed from the Evaluate Expression window in a C++ manner, as if they were local variables. To use a member variable in an expression you must use one of:

  — `this->member`
  — `this[0].member`
  — `*this.member`

  If the member variable is defined in a base class then `this->member` will return the correct results.

  In the Expression Evaluation window (and only there) you can access variables of a class by name. This means that `member` gives the same result as `this->member`. However, if you have more complex expressions such as:

  ```
  this->member1 * this->member2
  ```

  you cannot use:

  ```
  member1 * member2
  ```

- Base classes cannot be accessed in standard C++ notation. For example:

  ```
  class Base
  {
            char *name;
            char *A;
  };
  class Derived : public class Base
  {
            char *name;
            char *B;
            void do_sth();
  };
  ```

If you are in method `do_sth()` you can access the member variables `A`, `name`, and `B` through the `this` pointer. For example, `this->name` returns the name defined in class `Derived`.

To access `name` in class `Base`, the standard C++ notation is:

```
void Derived::do_sth()
{
          Base::name="value"; // sets name in the base class to “value”
}
```

However, the expression evaluation window does not accept `this->Base::name` because ADW does not understand the scope operator. You can access this value with:

```
this->::Base.name
```

- Though it is possible to call member functions in the form `Class::Member(...)`, this will give undefined results.
- **private**, **public**, and **protected** attributes are not recognized by the ADW Evaluate Expression window. This means that private and protected member variables can be accessed in the Evaluate Expression window because ADW treats them as public.

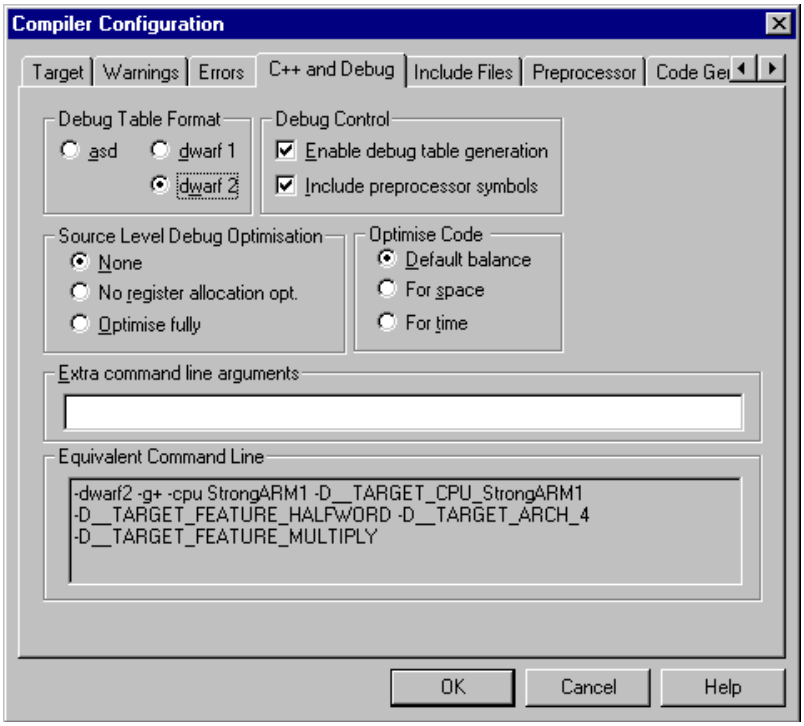
3.3 Debug Format Considerations

This section provides information about the debug table formats that can be generated by the ARM C++ compilers. It also describes how to change the format of the debug tables generated.

3.3.1 The debug table format

The ARM C++ compiler provides a number of options for building debug images. You can use the Compiler Configuration window in APM to set these options. Figure 3-10 shows an example of the Compiler Configuration window. Refer to *Configuring the compiler* on page 2-5 for more information on how to set compiler options for C++.

Figure 3-10: Compiler configuration window



By default, the C++ compiler produces DWARF2 format debug tables. The available formats are:

dwarf 2 This is the default format produced by APM for C++ projects. You should use this format unless you have specific reasons for using DWARF1.

dwarf 1 This is the DWARF version supported by SDT 2.11. You should use this format only if you have specific reasons for doing so. For example, you may wish to use a debugger that does not support DWARF2.

asd Do not use this format for C++. The ASD format cannot represent some C++ constructs, such as pointers to member functions. Using ASD will produce unpredictable results.

DWARF1 limitations

The DWARF1 debug table format has limitations that introduce severe restrictions to debugging C++ code. These include:

- DWARF1 provides no support for `#include` files. Stepping into member functions defined in `#include` files, and setting breakpoints on such functions, results in incorrect behavior.
- DWARF1 is less descriptive than DWARF2, and therefore has limited potential for building optimized debug images and objects.
- DWARF1 produces a much larger debug table than DWARF2. As a result, DWARF1 images can be significantly slower to load than DWARF2 images.

For these reasons, it is recommended that you use the DWARF2 debug table format.

Chapter 4

Using the ARM C++ Compilers

This chapter gives the information you need in order to make effective use of ARM C++. It assumes that you are familiar with the basic concepts of using command-line development tools, such as those provided with the ARM Software Development Toolkit. For an introduction to command-line development, see the ARM Software Development Toolkit Version 2.11 *User Guide* (ARM DUI 0040C).

This chapter contains the following sections:

4.1 About the ARM C++ Compilers

The ARM C++ compilers compile C++ that conforms to the ISO/IEC Draft Standard for C++ (December 1996). In addition, the ARM C++ compilers compile both ANSI C and the dialect of C used by Berkeley UNIX. Wherever possible, the compilers adopt widely used command-line options that are familiar to users of both UNIX and MS-DOS.

4.1.1 Compiler variants

There are two variants of the C++ compiler:

armcpp compiles C++ source into 32-bit ARM code

tcpp compiles C++ source into 16-bit Thumb code.

Throughout this chapter, armcpp and tcpp are referred to together as the ARM C++ compilers. Both compilers accept the same basic command-line options and the descriptions in this chapter apply to both. Where tcpp has added features or restrictions they are dealt with in Thumb-specific sections.

4.1.2 Source language modes

The ARM C++ compilers have a number of distinct source language modes that can be used to compile several varieties of C and C++ source code. The source language modes supported are:

C++ mode In C++ mode, the ARM C++ compilers compile C++ as defined by the ISO/IEC Draft Standard. The compilers have been tested against Suite++, *The Plum Hall Validation Suite for C++*, version 4.00. This is the default language mode.

Cfront mode In Cfront mode, the ARM C++ compilers are more likely to accept programs that Cfront accepts.

ANSI C mode In ANSI C mode, the ARM C++ compilers have been tested against release 7.00 of the Plum-Hall C Validation Suite (CVS). This suite has been adopted by the British Standards Institute for C compiler validation in Europe.

The compiler behavior differs from the behavior described in the language conformance sections of the CVS in the following ways:

- An empty initializer for an aggregate of complete type is a warning unless `-fussy` is specified. For example:

  ```
  int x[3] = {};
  ```

- The expression `sin(DBL_MAX)` causes a floating point trap if `-apcs /hardfp` is specified.
- There is no support for the `wctype.h` and `wchar.h` headers.

PCC mode In PCC mode, the compilers accept the dialect of C that conforms to the conventions of Berkeley UNIX (BSD 4.2) Portable C compiler C (PCC). This dialect is based on the original Kernighan and Ritchie definition of C, and is the one used on Berkeley UNIX systems.

For more information on how to use compiler options to set the source mode for the compiler, refer to *Setting the source language* on page 4-11.

4.1.3 Compatibility with ARM and Thumb C

The code generated by armcpp is completely compatible with that generated by armcc. Similarly, the code generated by tcpp is completely compatible with that generated by tcc.

Note *If you want to link compiled ARM and Thumb code together, refer to the SDT 2.11 User Guide.*

4.1.4 Library support

ARM C++ uses two libraries to support the compilation of Draft Standard C++ and ANSI C:

The Standard C library (armlib)

ARM C++ uses the Standard C library from the Software Development Toolkit 2.11 C compiler.

The ARM C++ library (armcpplib)

This library provides:

- The Rogue Wave Standard C++ library version1.2.1. This contains most of the standard library functions defined in the January 1996 Draft Standard.
- Helper functions for the C++ compiler, and a number of additional C++ functions not supported by the Rogue Wave library.

By default, these libraries are installed in the `/lib` directory of your SDT 2.11 installation directory. The accompanying header files are installed in the `/include` directory. The following sections describe the libraries in more detail.

Using the Standard C library

If you use only the ANSI C headers and your own headers, you will need only the following library support:

- C++ compiler helper functions. You can build a runtime version of the C++ helper functions from `/rebuild/cpplib/runtime.c`

  Enter: `armcpp -apcs` *`your_APCS_variant`* `-fc -DALL runtime.c` to build the `runtime.o` object file.

  Alternatively, you can use one of the prebuilt ARM C++ library variants. See below for more information.

- An ANSI C library binary from your SDT 2.11 installation. These are located in `/lib/armlib.xxx`, where `xxx` is the suffix for the appropriate library file.

Using the ARM C++ library

The ARM C++ library is supplied in binary form in 26 pre-compiled variants. The ARM C++ library provides:

- The Rogue Wave Standard C++ library. This is supplied in binary form only, as part of the precompiled armcpplib variants, and as sublibraries for use when rebuilding armcpplib libraries. The object files are located in your SDT 2.11 `/rebuild/stdlib` directory.

  Refer to *The Standard C++ Library* on page 6-4 for more information on the C++ library implementation.

  Rogue Wave provides comprehensive documentation for the Standard C++ library in HTML format. This is installed in your SDT 2.11 `/HTML` directory.

- Additional library functions. These functions are built into the C++ library to support the compiler, and to provide basic support for some parts of the Standard C++ library that are not supported by the Rogue Wave implementation. Source code for the additional functions is located in `/lib/rebuild/cpplib`.

  The additional functions are:

  — C++ compiler helper functions. Calls to these functions are generated by the compiler to implement certain language constructs.
  — Basic support for exceptions. The header file `exception.h` is installed in your SDT 2.11 `/include` directory. If you are using the Rogue Wave Standard C++ library header `exception`, you do not need `exception.h`
  — Support for the `new` operator. The header file `new.h` is installed in your SDT 2.11 `/include` directory. If you are using the Rogue Wave Standard C++ library header `new`, you do not need `new.h`
  — Partial implementation of the `type_info` class. The header file `typeinfo` is installed in your SDT 2.11 `/include` directory.

Partial support for C++ iostreams. The header file `iostream.h` is installed in your SDT 2.11 `/include` directory. In addition, an identical header file `stream.h` is installed in `/include` for compatibility with previous (beta) releases of ARM C++.

Refer to *Rebuilding the ARM C++ Library* on page 6-5 for information on rebuilding armcpplib.

4.2 File Usage

This section describes:

- the file naming conventions used by ARM C++
- how ARM C++ searches for `#include` header and source files.

4.2.1 Naming conventions

ARM C++ uses suffix naming conventions to identify the classes of file involved in the compilation and linking processes. The names used by the ARM C++ on the command-line, and as arguments to the C++ preprocessor `#include` directive, map directly to host filenames under UNIX and MS-DOS.

ARM C++ recognizes the following file suffixes:

`filename``.c` A C++ or C source file.

In addition, ARM C++ recognizes suffixes of the form `.c*`, such as:

- `.cpp`
- `.cp`
- `.c++`
- `.cc`

and their uppercase equivalents.

`filename``.h` A header file. This is a convention only. It is not specifically recognized by the compiler.

`filename``.o` An ARM object file.

`filename``.s` An ARM or Thumb assembly language file.

`filename``.lst` A compiler listing file. This is an output file only.

Portability

ARM C++ supports the use of multiple file naming conventions on any host. Follow these guidelines:

- ensure that filenames do not contain spaces
- make embedded pathnames relative, rather than absolute.

In each host environment, ARM C++ supports:

- Native filenames.
- Pseudo UNIX filenames. These have the format:
  `host-volume-name:/rest-of-unix-file-name`
- UNIX filenames.

Filenames are parsed as follows:

- a name starting with `host-volume-name:/` is a pseudo UNIX filename
- a name containing `/` is a UNIX filename
- otherwise, the name is a host name.

Filename validity

The compiler does not check that filenames are acceptable to the host filing system. If a filename is not acceptable, the compiler reports that the file could not be opened, but gives no further diagnosis.

Output files

By default, the object, assembler, and listing files created by the compiler are stored in the current directory. Object files are written in ARM Object Format (AOF). AOF is described in the SDT 2.11*Reference Guide*.

4.2.2 Included files

There are a number of factors that affect how the ARM C++ compilers search for included header and source files, including:

- The `-I` and `-j` compiler options.
- The `-fk` and `-fd` compiler options.
- The value of the environment variable *`ARMINC`*.
- Whether the filename is an absolute filename or a relative filename.
- Whether the filename is between angle brackets or double quotes. This determines whether or not the file is sought in the in-memory filing system.

The in-memory filing system (:mem)

Like ARM C, ARM C++ has the ANSI C library headers built into a special, textually compressed, in-memory filing system. By default, the C header files are used from this filing system. The in-memory filing system can be specified explicitly on the command-line as `:mem`.

Enclosing an `#include` filename in angle brackets indicates that the included file is a system file and ensures that the compiler looks first in its built-in filing system. For example:

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

Enclosing an `#include` filename in double quotes in the `#include` directive indicates that it is not a system file. For example:

```
#include "myfile.h"
```

In this example, the compiler looks for the specified file in the appropriate search path. Refer to *Specifying search paths* on page 4-13 for detailed information on how ARM C++ searches for include files.

Note *The ARM C++ header files are not stored in :mem. The compiler will search the remainder of the search path for C++ system include files enclosed in angle brackets.*

The current place

By default, ARM C++ adopts the search rules used by Berkeley UNIX systems. Under these rules, source files and `#include` header files are searched for relative to the *current place*. The current place is the directory containing the source or header file currently being processed by the compiler.

When a file is found relative to an element of the search path, the name of the directory containing that file becomes the new current place. When the compiler has finished processing that file, it restores the previous current place. At each instant, there is a stack of current places corresponding to the stack of nested `#include` directives.

For example, if the current place is `/ARM211/include` and the compiler is seeking the include file `sys/defs.h`, it will locate `/ARM211/include/sys/defs.h` if it exists.

When the compiler begins to process `defs.h`, the current place becomes `/ARM211/include/sys`. Any file included by `defs.h` that is not specified with an absolute pathname is sought relative to `/ARM211/include/sys`.

You can disable the stacking of current places with the compiler option `-fk`. This option makes the compiler use the search rule originally described by Kernighan and Ritchie in *The C Programming Language*. Under this rule, each non-rooted user `#include` is sought relative to the directory containing the source file that is being compiled.

The ARMINC environment variable

You can set the *`ARMINC`* environment variable to specify a list of directories to be searched for included header and source files. Directories listed here will be searched immediately after any directories specified by the `-I` option on the command-line. If the `-j` option is used, *`ARMINC`* is ignored.

The search path

Table 4-1 shows how the various command-line options affect the search path used by the compiler for included header and source files. The search path is different for double quoted include files and angle bracketed include files. The following conventions are used in the table:

- `:mem` means the in-memory file system in which ARM C++ stores ANSI C header files. See *The in-memory filing system (:mem)* on page 4-6 for more information.
- `ARMINC` is the list of directories specified by the *`ARMINC`* environment variable, if it is set.
- `CP` is the current place. See *The current place* on page 4-6 for more information.
- `Idir` and `jdirs` are the directories specified by the `-I` and `-j` compiler options. Note that multiple `-I` options may be specified and that directories specified by `-I` are searched prior to directories specified by `-j`, irrespective of their relative order on the command-line.

Table 4-1: Include file search paths

Compiler Option	<include>	“include.h”
not -I or -j	:mem, ARMINC, :mem	CP, ARMINC,:mem
-j	jdirs, :mem	CP, jdirs, :mem
-I	mem, Idirs, ARMINC, :mem	CP, Idirs, ARMINC, :mem
both -I and -j	Idirs, jdirs, :mem	CP, Idirs, jdirs, :mem
-fd	no effect	Removes CP from the search path. Double quoted include files are searched for in the same way as angle bracketed include files.
-fk	no effect	Makes CP follow Kernighan and Ritchie search rules.

4.3 Command Syntax

This section describes the command syntax for the ARM C++ compilers.

Many aspects of compiler operation can be controlled using command-line options. All options are prefixed by a minus sign, and some options are followed by an argument. Whenever this is the case, the ARM C++ compilers allow space between the option letter and the argument.

Note *This is not always true of other C++ compilers, so the following descriptions show the form that would be acceptable to most UNIX C++ compilers. They also show the case of the letter that would be accepted by a UNIX C++ compiler.*

The command-line options are divided into the following subsections. Options that control related aspects of compiler operation are grouped together. The major groups are:

4.3.1 Invoking the compiler

The command for invoking the ARM C++ compilers is one of:

```
armcpp options sourcefile(s)
```

```
tcpp options sourcefile(s)
```

where *options* are zero or more of the compiler options described in the sections below, and *sourcefile(s)* are the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files, and creates object, assembler, and listing files in the current directory.

The following points apply to specifying compiler options:

- Compiler options beginning with `-W` (warning options), `-E` (error options), and `-f` can have multiple modifiers specified. For example, `-fh` and `-ff` can be combined as `-ffh`. Similarly `-Wf` and `-Wg` can be combined as `-Wfg`.
- Other compiler options, such as debug options, may have specific shortcuts for common combinations. These are described in the appropriate sections below.

Reading compiler options from a file

Use the `-via filename` compiler option to open a file and read additional command-line options from it. For example:

```
armcpp -via input.txt options source.c
```

Specifying keyboard input

Use `-` (minus) as the source filename to instruct the compiler to take input from the keyboard. Input is terminated by entering `Ctrl-D` under UNIX, or `Ctrl-Z` under MS-DOS. If:

- no output file is specified
- the -E compiler option is not specified

an assembly listing for the function is sent to the output stream at the end of each C++ function.

Getting help

Use the `-help` compiler option to view a summary of the compiler command-line options.

Redirecting standard errors

Use the `-errors filename` compiler option to redirect compiler error output to a file.

4.3.2 APCS command-line options

Use the following command-line options to specify the variant of the ARM Procedure Call Standard (APCS) that is to be used by the compiler:

```
-apcs [3]qualifiers
```

Refer to Chapter 5 of the SDT 2.11 *Reference Guide* for more information on the ARM Procedure Call Standard. The following rules apply to the APCS command-line option:

- There must be a space between `-apcs` and the first qualifier
- At least one qualifier must be present, and there must be no space between qualifiers.

The qualifiers are listed below.

APCS variants

`/32bit` 32-bit APCS variant. This option is not available for Thumb.

`/reentrant` Re-entrant APCS variant. This option is not available for Thumb.

`/nonreentrant` Non re-entrant APCS variant. This option is not available for Thumb.

Stack checking

`/swstackcheck`

Software stack-checking APCS variant. This is the default for ARM.

`/noswstackcheck`

No software stack-checking APCS variant. This is the default for Thumb.

Frame pointers

`/fp` Use a dedicated frame-pointer register. This option is not available for Thumb.

`/nofp` Do not use a frame-pointer register. This always applies to Thumb.

Floating-point compatibility

`/fpe2` Floating-point emulator 2 compatibility. This option is not available for Thumb.

`/fpe3` Floating-point emulator 3 compatibility. This option is not available for Thumb.

`/fpregargs` Floating-point arguments are passed in floating-point registers. This option is not available for Thumb.

`/nofpregargs` Floating-point arguments are not passed in floating-point registers. This option is not available for Thumb.

`/softfp` Call software floating-point library functions to emulate floating-point code. This is the default for ARM and the only floating-point method available for Thumb. Note that `/softfp` implies `/softfloats` and `/softdoubles`. This option is not available for Thumb.

`/hardfp` Generate ARM coprocessor instructions for floating-point. You may also specify `/fpe2` or `/fpe3` and `/fpr` or `/nofpr`. Note that `/hardfp` implies not `/softfp`. This option is not available for Thumb.

`/softdoubles` Call software floating-point emulation library functions for code that uses **double**. You can use this option in combination with `/hardfp` to use software emulation for **double** and hardware for **float**. If used in combination with `/softfloats` it is equivalent to `/softfp`. This option is not available for Thumb.

`/softfloats` Call software floating-point emulation library functions for code that uses **float**. You can use this option in combination with `/hardfp` to use software emulation for **float** and hardware for **double**. If used in combination with `/softdoubles` it is equivalent to `/softfp`. This option is not available for Thumb.

ARM/Thumb interworking

`/interwork` Compile code for ARM/Thumb interworking. See the SDT 2.11 *User Guide* for more information on ARM/Thumb interworking.

`/nointerwork` Do not compile code that is suitable for ARM/Thumb interworking. This is the default.

Narrow parameters

`/wide` For functions with parameters of narrow type (**char**, **short**, **float**), this option applies the default argument promotions to its corresponding actual arguments, passing them as **int** or **double**. This is known as callee-narrowing, and is the default.

`/narrow` For functions with parameters of narrow type (**char**, **short**, **float**), this option converts the corresponding actual arguments to the type of the parameter. This is known as caller-narrowing. It requires that all calls be within the scope of a declaration containing the function prototype.

4.3.3 Setting the source language

The following options can be used to specify the source language that the compiler is to accept, and how strictly it enforces the standards or conventions of that language. If no source language option is specified, the compiler attempts to compile C++ that conforms to the Draft Standard.

`-ansi` Compiles ANSI standard C.

`-ansic` Compiles ANSI standard C. This option is synonymous with the `-ansi` option.

`-cfront` The compiler alters its behavior so that it is more likely to accept C++ programs that Cfront accepts. For example, in Cfront mode the scope of identifiers declared in **for** statements extends to the end of the statement or block containing the **for** statement.

The following differences change the way in which C++ programs that conform to the Draft Standard behave:

- In Cfront mode `D::D` in the following example is a copy constructor and `D::operator=` is a copy assignment:

  ```
  struct B { };
  struct D { D(const B&); operator=(const B&); };
  ```

  In non-Cfront mode, the compiler generates an implicit copy constructor and copy assignment. A warning is generated. In C++ mode, only `'const D&'` or `'D&'` will work.

- Implicit conversions to **void****, **void*****, and so on, are allowed. For example:

  ```
  int* p;
  void **pp = &p;
  ```

  This is not allowed in C or C++. No warning is given.

- The scope of variables declared in **for** statements extends to the end of the enclosing scope. For example:

  ```
  for (int i = 0; i < 10; ++i) g(i);
  int j = i;               // accepted only in Cfront mode
  ```

  No warning is given.

- The macro `__CFRONT_LIKE` is predefined to `1`.

The following differences allow programs that do not conform to the Draft Standard to compile:

- Typedefs are allowed in some places where only true classnames are permitted by the Draft Standard. For example:

  ```
  struct T { T(); ~T(); };
  typedef T X;
  X::X() { }                              // not legal
  void f(T* p) { p->X::~X(); }            // not legal
  ```

  No warning is given.

- The following constructs generate warnings instead of errors:
  - jumps past initializations of objects with no constructor
  - `delete [e] p`
  - `enum {,k}`
  - `enum {k2,}`
  - `class T { friend X; };  // should be friend class X`

`-fc` Enables limited PCC mode. This mode allows you to use PCC-style header files in an otherwise strict ANSI C or C++ environment. This mode:

- supports system programming in otherwise strict ANSI C or C++ environment
- allows you to use libraries of functions implemented in old-style C from an application written in ANSI C or C++ that conforms to the Draft Standard.

The limited PCC option:

- Allows characters after `#else` and `#endif` preprocessor directives. The characters are ignored and warnings are generated.
- Suppresses warnings about explicit casts of integers to function pointers.
- Permits the dollar character in identifiers. Linker-generated symbols often contain "$$", and all external symbols containing "$$" are reserved to the linker.

`-fussy` Is extra strict about enforcing conformance to the Draft C++ standard, the ANSI C standard, or PCC conventions. For example,in C++ mode the following code gives an error when compiled with `-fussy` and a warning without:

```
static struct T {int i; };
```

Because no object is declared, the **static** is spurious. In a strict reading of the Draft Standard, it is illegal.

`-pcc` Compiles (BSD 4.2) Portable C compiler C. This dialect is based on the original Kernighan and Ritchie definition of C, and is the one used to build UNIX systems. The `-pcc` option alters the language accepted by the compiler, but the built-in ANSI C headers are still used.

`-pedantic` This is a synonym for `-fussy`.

`-strict` This is a synonym for `-fussy`.

`-fy` Treats enumerations as signed integers. This option is off by default (no forced integers).

`-zc` Make **char** signed. It is normally unsigned in C++ and ANSI C modes, and signed in PCC mode.

4.3.4 Specifying search paths

The following options allow you to specify the directories that are searched for included files. The precise search path will vary depending on the combination of options selected, and whether the include file is enclosed in angle brackets or double quotes. Refer to the section *Included files* on page 4-6 for full details of how these options work together.

`-I`*`dir-name`*
: Adds the specified directory to the list of places that are searched for included files. The directories are searched in the order they are given, by multiple `-I` options.The in-memory filing system is specified by `:mem`.

`-fk`
: Uses Kernighan and Ritchie search rules for locating included files. The current place is defined by the original source file and is not stacked. See *The current place* on page 4-6 for more information. If you do not use this option, Berkeley-style searching is used.

`-fd`
: Makes the handling of quoted include files the same as angle-bracketed include files. Specifically, the current place is excluded from the search path.

`-j`*`dir-list`*
: Adds the specified comma-separated list of directories to the end of the search path, after all directories specified by `-I` options. It also forces the compiler to search the in-memory filing system after all other searches have failed. You cannot have more than one `-j` option on a command-line.

4.3.5 Setting preprocessor options

The following command-line options control aspects of the preprocessor. Refer to *Pragmas controlling the preprocessor* on page 4-28 for descriptions of other preprocessor options that can be set by pragmas.

`-E` Executes only the preprocessor phase of the compiler. The output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard UNIX/MS-DOS notation. For example:

```
toolname -E something.c > rawc
```

where *`toolname`* is either `armcpp` or `tcpp`. By default, comments are stripped from the output. See also the `-C` option, below.

`-C` Retains comments in preprocessor output when used in conjunction with `-E`. Note that this option is different from the `-c` (lowercase) option, that suppresses the link step. See *Specifying output format* on page 4-15 for a description of the `-c` option.

`-M` Executes only the preprocessor phase of the compiler, as with `-E`. However, the only output produced is a list, on the standard output stream, of makefile dependency lines suitable for use by a make utility. This can be redirected to a file using standard UNIX/MS-DOS notation. For example:

```
toolname -M something.c >> Makefile
```

where *`toolname`* is either `armcpp` or `tcpp`.

`-Dsymbol=value`

Defines *`symbol`* as a preprocessor macro, as if the following line were at the head of the source file:

```
#define symbol value
```

This option can be repeated.

`-Dsymbol` Defines *`symbol`* as a preprocessor macro, as if the following line were at the head of the source file:

```
#define symbol
```

The symbol is given the default value 1. This option can be repeated.

`-Usymbol` Undefines *`symbol`*, as if the following line were at the head of the source file:

```
#undef symbol
```

This option can be repeated.

4.3.6 Specifying output format

By default, source files are compiled and linked into an executable image. The following options can be used to direct the compiler to create unlinked object files, assembly language files, and listing files from C or C++ source files. You can also specify the output directory for files created by the compiler. Refer to *Setting preprocessor options* on page 4-14 for information on creating listings from the preprocessor output.

`-c`
: Does not perform the link step. The compiler compiles the source program, and leaves the object files in either the current directory, or the output directory if specified by the `-o` option. Note that this option is different from the `-C` (uppercase) option described in *Setting preprocessor options* on page 4-14.

`-fi`
: When used with `-list`, lists the lines from any files included with directives of the form `#include "`*`file`*`"`.

`-fj`
: When used with `-list`, lists the lines from any files included with directives of the form `#include <`*`file`*`>`.

`-fu`
: When used with `-list`, lists unexpanded source. By default, if `-list` is specified, the compiler lists the source text as seen by the compiler after preprocessing. If `-fu` is specified, the unexpanded source text is listed. For example:

```
p = NULL; /* assume #defined NULL (0) */
```

By default, this is listed as `p = (0)`. With `-fu` specified it is listed as `p = NULL`.

`-list`
: Creates a listing file. This consists of lines of source interleaved with error and warning messages. You can gain finer control over the contents of this file by using the `-fi`, `-fj`, and `-fu` options in combination with `-list`.

`-o` *`file`*
: Names the file that holds the final output of the compilation step.

- In conjunction with `-c`, it gives the name of the object file.
- In conjunction with `-S`, it gives the name of the assembly language file.
- If neither `-c` or `-S` are specified, it names the final output of the link step.

If no `-o` option is specified, the name of the output file defaults to the name of the input file with the appropriate filename extension. For example, the output from `file1.c` is named `file1.o` if the `-c` option is specified, and `file1.s` if `-S` is specified.

If neither `-c` or `-S` is specified, an executable image called `file2` is created.

If *`file`* is specified as `-`, output is sent to `stdout`.

`-S`
: Writes a listing of the assembly language generated by the compiler to a file, and does not generate object code. The name of the output file defaults to `file.s` in the current directory, where `file.c` is the name of the source file stripped of any leading directory names. The default can be overridden using the `-o` option.

4.3.7 Specifying the target processor and architecture

The options described below specify the target processor or architecture for a compilation. The compiler may take advantage of certain features of the selected processor or architecture. This may make the code incompatible with other ARM processors. For example, some processors cannot use halfword instructions.

The following general points apply to processor and architecture options:

- If you specify an `-architecture` option, the compiler compiles code that runs on any processor that supports that architecture.
- If you specify a `-processor` option, the compiler compiles code specifically for that processor. The code may be incompatible with other ARM processors that support the same architecture.
- To enable halfword load and store instructions, specify option `-architecture 4` or `-architecture 4T`, or specify an appropriate processor using the `-processor` option.
- Specifying a Thumb-aware processor, such as `-processor ARM7TM`, to armcpp does not make armcpp generate Thumb code. Instead, it generates ARM code that uses the Architecture 4 halfword load and store ARM instructions. This option is not available with tcpp, because tcpp always generates Thumb code.

Thumb None of the following options are available for Thumb.

The following options are available for ARM:

`-architecture` *n*

Specifies the ARM architecture version that compiled code will comply with. Valid values for *n* are:

- 2
- 3
- 3M
- 4
- 4T

`-cpu` *name* This is a synonym for the `-processor` option.

`-fpu` *name* Select the target FPU, where *name* is one of:

- amp - Attached Media Processor
- fpa - Floating Point Accellerator

`-processor` *name*

Compiles code for the specified ARM processor where *name* is the name of the ARM processor. Valid values are:

ARM2	Supports architecture 2.
ARM3	Supports architecture 2.
ARM6	Supports architecture 3. This is the default if no `-processor` and no `-architecture` options are specified. If `-architecture` is specified and `-processor` is not, the default processor is generic.

ARM7	Supports architecture 3.
ARM7M	Supports architecture 3M.
ARM7TM	Supports architecture 4T.
ARM8	Supports architecture 4.
StrongARM1	Supports architecture 4.
SA1500	Supports architecture 4. Selecting this processor option implies `-fpu amp` and `-APCS /softdoubles`.
generic	This processor name is used when compiling for a specified architecture and no processor is specified.

4.3.8 Generating debug information

Use the following options to specify whether debug tables are generated for the current compilation, and the format of the debug table to be generated. Refer to *Debug Format Considerations* on page 3-11 for more information on debug table formats.

Note that the effect of:

```
-g+ -gt<T-options> -gx<X-options>
```

can be combined in a single command-line argument:

```
-g<X-options><T-options>
```

The following debug options are available:

`-asd` Use ASD debug table format. This option is not recommended for C++.

`-dwarf` Use DWARF1 debug table format. This option is not recommended for C++.

`-dwarf1` Use DWARF1 debug table format. This option is not recommended for C++.

`-dwarf2` Use DWARF2 debug table format. This is the default for C++.

`-g+` Switches the generation of debug tables on for the current compilation. Debug table options are as specified by `-gt` and `-gx`.

`-g-` Switches the generation of debug tables off for the current compilation.

`-gt`*letters* Specifies the debug tables entries that generate source level objects. Debug tables can be very large, so it can be useful to limit what is included.

- `-gt` All available entries should be generated.
- `-gtp` Tables should not include preprocessor macro definitions. This option is ignored if DWARF1 debug tables are generated, as there is then no way to describe macros.

`-gx`*letters* Specifies the level of optimization allowed when generating debug tables:

- `-gx` No optimizations
- `-gxr` Unoptimized register allocation
- `-gxo` Full optimization. Note the following points:
  - You must take care with the values of local variables, as many local variables may occupy the same register and you might get unexpected values displayed in the debugger.
  - The code generated with `-gxo` in force is degraded compared with that generated when no debug tables are generated. Some optimizations normally performed by the compiler must be disabled because they cannot be described within the debug table formats.

4.3.9 Controlling code generation

The following options allow you to control various aspects of the code generated by the compiler, such as optimization, use of the code and data areas, endianness, and alignment. Refer to *Pragmas* on page 4-27 for information on additional code generation options that are controlled using pragmas.

Controlling optimization

`-Ospace` Optimizes to reduce image size at the expense of increased execution time.

`-Otime` Optimizes to reduce execution time at the expense of a larger image.

`-zi`*`Number`* Defines the maximum number of instructions allowed to generate an integer literal inline before using `LDR rx,=`*`value`* where *`number`* is between 1 and 4. The default is 2.

Controlling code and data areas

`-ff` Does not embed function names in the code area. This option is enabled by default to reduce the size of the code area. See also the `-fn` option below.

In general, it is not useful to specify `-ff` with `-apcs /nofp`. See *APCS command-line options* on page 4-9 for more information. This option is disabled by default.

`-fn` Embeds function names in the code area. This improves the readability of the output produced by the stack backtrace runtime support function and the `_mapstore()` function. However, it increases the size of the code area by approximately 5% for C programs, and more for C++ programs. See also the `-ff` option above.

`-fw` Allows string literals to be writable, as expected by some UNIX code, by allocating them in the program data area rather than the notionally read-only code area. Note that this also stops the compiler reusing a multiple-occurring string literal.

`-zo` Generates one AOF area for each function.

`-zt` Disallows tentative declarations. If this option is specified, the compiler assumes that any occurrence of a top level construct such as `int i;` is a definition without initializer, rather than a tentative definition. Any subsequent definition with initializer in the same scope will generate an error.

Note that this option has effect only for C, not for C++. This option is useful in combination with the `-zz` and `-zas` options.

`-zz`*`ZI_Threshold_Size`*

Sets the value of the zero-initialized data threshold size. The compiler will place uninitialized global variables in the Zero Initialized (ZI) data area if their size is greater than *`ZI_Threshold_Size`* bytes in the ZI data area. For example, you can force uninitialized variables of any size to be placed in the ZI data area by specifying `-zz0`, though this may significantly increase your code size.

To avoid increased code size, use this option in combination with `-zt`. Option `-zzt` provides a convenient shorthand. The default threshold size is 8 bytes.

`-zzt`*`ZI_Threshold_Size`*

Combines the `-zt` and `-zz` options. For example, specify `-zzt0` to force the compiler to disallow tentative declarations and place all uninitialized variables in the ZI data area.

Setting endianness

`-bigend` Compiles code for an ARM operating with big-endian memory. The most significant byte has lowest address.

`-littleend` Compiles code for an ARM operating with little-endian memory. The least significant byte has lowest address. This is the default.

Controlling SWI calling standards

`-fz` Instructs the compiler that an inline SWI may overwrite the contents of the link register. This option is usually used for modules that run in Supervisor mode, and that contain inline SWIs.

Load and store options

`-za`*`Number`* Specifies whether LDR may only access word-aligned addresses. Valid values are:

- `-za1` LDR may only access word-aligned addresses. This is the default.
- `-za0` LDR is not restricted to accessing word-aligned addresses.

`-zr`*`Number`* Limits the number of register values transferred by load multiple and store multiple instructions generated by the compiler to *`Number`*. Valid values for *`Number`* are 3 to 16 inclusively. The default value is 16.

This option can be used to reduce interrupt latency. Note that inline assembler is not subject to the limit imposed by the `-zr` option.

The Thumb compiler does not support this option.

`-zap`*`Number`* Specifies whether pointers to structures are assumed to be aligned on at least *`struct minimal alignment`* boundaries, as set by `-zas`. Valid values are:

- `-zap1` Pointers to structures are assumed to be aligned on at least `<struct minimal alignment>` boundaries. This is the default.
- `-zap0` Pointers to structures are not assumed to be aligned on at least *`struct minimal alignment`* boundaries. Casting `short[ ]` to `struct {short, short,...}` does not cause a problem.

Alignment options

`-zas`*`Number`* Specifies the minimum byte alignment for structures. Valid values for *`Number`* are:

`1, 2, 4, 8`

The default is 4 for both armcpp and tcpp. This allows structure copying to be implemented more efficiently by copying in units of words, rather than bytes. Setting a lower value reduces the amount of padding required, at the expense of the speed of structure copying.

`-zat`*`Number`* Specifies the minimum byte alignment for top-level static objects, such as global variables. Valid values for *`Number`* are:

`1, 2, 4, 8`

The default is 4 for armcpp and 1 for tcpp.

4.3.10 Controlling warning messages

The compiler issues warnings to indicate potential portability problems or other hazards. The compiler options described below allow you to turn specific warnings off. For example, you may wish to turn warnings off if you are in the early stages of porting a program written in old-style C. The options are on by default, unless specified otherwise. See also *Specifying additional checks* on page 4-24 for descriptions of additional warning messages.

The general form of the `-W` compiler option is:

`-W[`*`options`*`][+][`*`options`*`]`

where *`options`* are one or more characters.

If the `+` character is included in the characters following the `-W`, the warnings corresponding to any following letters are enabled rather than suppressed.

You can specify multiple options. For example:

```
-Wad+fg
```

turns off the warning messages specified by `a`, `d`, and turns on the warning message specified by `f` and `g`.

The warning message options are as follows:

`-W` Suppresses all warnings. If one or more letters follow the option, only the warnings controlled by those letters are suppressed.

`-Wa` Suppresses the warning message:

```
Use of the assignment operator in a condition context
```

This warning is given when the compiler encounters a statement such as:

```
if (a = b) {...
```

where it is possible that:

```
if ((a = b) != 0) {...
```

was intended, or that:

```
if (a == b) {...
```

was intended. This warning is suppressed by default in PCC mode.

`-Wd` Suppresses the warning message:

```
Deprecated declaration foo() - give arg types
```

This warning is given when a declaration without argument types is encountered in ANSI C mode. This warning is suppressed by default in PCC mode.

In ANSI C, declarations like this are deprecated. However, it is sometimes useful to suppress this warning when porting old code.

In C++, `void foo();` means `void foo(void);` and no warning is generated.

`-Wf` Suppresses the message:

```
Inventing extern int foo()
```

This is an error in C++ and a warning in ANSI C. Suppressing this may be useful when compiling old-style C in ANSI C mode. This warning is suppressed by default in PCC mode.

`-Wg` Suppresses the warning given if an unguarded header file is `#included`. This warning is off by default. It can be enabled with `-W+g`. An unguarded header file is a header file not wrapped in a declaration such as:

```
#ifdef foo_h
#define foo_h
/* body of include file */
#endif
```

`-Wi` Suppresses the implicit constructor warning. This warning applies to C++ only. It is issued when the code requires a constructor to be invoked implicitly. For example:

```
struct X { X(int); };
X x = 10;                       // actually means, X x = X(10);
                                // See the Annotated C++
                                // Reference Manual p.272
```

This warning is off by default. It can be enabled with `-W+i`.

`-Wl` Lower precision in wider context. This warning arises in cases like:

```
long x; int y, z; x = y*z
```

where the multiplication yields an **int** result that is then widened to **long**.This warns in cases where the destination is **long long**, and in cases of portability problems to targets with 16-bit integers or 64-bit longs. This option is off by default. It can be enabled with `-W+l`.

`-Wn` Suppresses the warning message:

```
Implicit narrowing cast
```

This warning is issued when the compiler detects the implicit narrowing of a long expression in an **int** or **char** context, or the implicit narrowing of a floating-point expression in an integer or narrower floating-point context.

Such implicit narrowings are almost always a source of problems when moving code that has been developed on a fully 32-bit system (such as ARM C++) to a system in which integers occupy 16 bits and longs occupy 32 bits. This is suppressed by default.

`-Wp` Suppresses the warning message:

```
non-ANSI #include <...>
```

The ANSI C standard requires that you use `#include <...>` for ANSI C headers only. However, it is useful to disable this warning when compiling code not conforming to this aspect of the standard. This option is suppressed by default, unless the `-strict` option is specified.

`-Wr` Suppresses the implicit virtual warning. This warning is issued when a non-virtual member function of a derived class hides a virtual member of a parent class. It is applicable to C++ only. For example:

```
struct Base { virtual void f(); };
struct Derived : Base { void f(); };
                      // warning 'implicit virtual'
```

Adding the **virtual** keyword in the derived class avoids the warning.

`-Ws` Warns when the compiler inserts padding in a **struct**. This is warning is off by default. It can be enabled with `-W+s`.

`-Wt` Suppresses the unused 'this' warning. This warning is issued when the implicit 'this' argument is not used in a non-static member function. It is applicable to C++ only. The warning can be avoided by making the member function a static member function.

`-Wu` For C code, suppresses warnings about future compatibility with C++ for both armcpp and tcpp. This option is off by default. It can be enabled with `-W+u`.

`-Wv` Suppresses the warning message:

```
 Implicit return in non-void context
```

This is most often caused by a return from a function that was assumed to return **int**, because no other type was specified, but is being used as a void function. This is widespread in old-style C. This warning is suppressed by default in PCC mode.

This is always an error in C++.

`-Wz` This warning is off by default. When enabled with `-W+z`, a warning is displayed when the compiler detects code that performs structure assignment. For example:

```
struct x {int x,y}
struct x a,b; struct x foo();
a = b;                              // warning
a = foo();                          // warning
```

This may be useful for debugging purposes.

4.3.11 Specifying additional checks

The options described below control a variety of compiler features, including features that make some checks more rigorous than usual. These additional checks can be an aid to portability and good coding practice.

`-fa` Checks for certain types of data flow anomalies. The compiler performs data flow analysis as part of code generation. The checks enabled by this option indicate when an automatic variable could have been used before it has been assigned a value. The check is pessimistic and will sometimes report an anomaly where there is none, especially in code like this:

```
int initialized = 0, value;
...
if (initialized) { int v = value; ...
... value = ...; initialized = 1; }
```

Here, `value` is read-only if `initialized` has been set. This is a semantic deduction, not a data flow implication, so `-fa` reports an anomaly. In general, it is useful to check all code using `-fa` at some stage during its development.

`-fe` Checks that external names used within the file are unique when reduced to six case-insensitive characters. Some linkers support as few as six significant characters in external symbol names. This can cause problems if a system uses two names such as `getExpr1` and `getExpr2`, where only the eighth character is unique.

This check can be made only within a single compilation unit, so it cannot catch all such problems. This option is an aid to portability.

`-fh` Checks that all external objects are declared before use, and that all file-scoped static objects are used. If external objects are only declared in included header files and are never inline in a source file, these checks directly support good modular programming practices.

When writing production software, you are encouraged to use the `-fh` option in the later stages of program development. The extra diagnostics produced can be annoying in the earlier stages.

`-fp` Reports on explicit casts of integers to pointers, for example:

```
char *cp = (char *) anInteger;
```

This warning indicates potential portability problems.

Casting explicitly between pointers and integers, although not clean, is not harmful on the ARM processor where both are 32-bit types. Implicit casts are reported unless suppressed by the `-Wc` option. See *Controlling warning messages* on page 4-21 for more information on the `-Wc` option.

`-fv` Reports on all unused declarations, including those from standard headers.

`-fx` Enables all warnings that are suppressed by default.

4.3.12 Controlling error messages

The compiler issues errors to indicate that serious problems exist in the code it is attempting to compile.

The compiler options described below allow you to:

- turn specific recoverable errors off
- downgrade specific errors to warnings.

Note *These options force the compiler to accept C++ source that would normally produce errors. If you use any of these options, it means that the C++ source does not conform to the Draft Standard. These options may be useful during development, or when importing code from other environments.*

The general form of the `-E` compiler option is:

`-E[`*`options`*`][+][`*`options`*`]`

where *`options`* are one or more of the letters described below.

If the `+` character is included in the characters following the `-E`, the errors corresponding to any following letters are enabled rather than suppressed.

You can specify multiple options. For example:

```
-Eac
```

turns off the error messages specified by `a` and `c`

The following options are on by default unless specified otherwise:

`-Ea` Downgrades access control errors to warnings. For example:

```
class A { void f() {}; };  // private member
A a;
void g() { a.f(); }        // erroneous access
```

`-Ec` Suppresses all implicit cast errors, such as implicit casts of a non-zero **int** to **pointer**.

`-Ef` Suppresses errors for unclean casts, such as **short** to **pointer**.

`-Ei` Downgrades constructs of the following kind from errors to warnings. For example:

```
const i;
Error: declaration lacks type/storage-class (assuming 'int'): 'i'
```

This option applies to C++ only.

`-El` Suppresses errors about linkage disagreements where functions are implicitly declared **extern** and later defined as **static**. This option applies to C++ only.

`-Ep` Suppresses the error that occurs if there are extraneous characters at the end of a preprocessor line. This error is suppressed by default in PCC mode.

`-Ez` Suppresses the error that occurs if a zero-length array is used.

4.3.13 Miscellaneous compiler features

The following compiler options control miscellaneous features of the compiler.

`-zi`*`File`*
: Use a preincluded header file, where *`File`* is the name of a header file that is to be included before compilation starts.

`-zp`*`LetterDigit`*
: Emulates `#pragma` directives. The letter and digit that follow it are the same characters that would follow the "`-`" of a short form `#pragma` directive. See *Pragmas* on page 4-27 for details.

4.4 Other Compiler Features

This section describes:

- pragmas
- function declaration keywords
- variable declaration keywords
- type qualifiers.

4.4.1 Pragmas

Pragmas are not portable to other compilers. Pragmas are recognized by the compiler in two forms:

short form `#pragma -LetterDigit`

long form `#pragma [no_]feature-name`

A short-form pragma given without a digit resets that pragma to its default state. Otherwise, the pragma is set to the state specified. For example:

```
#pragma -s1
#pragma no_check_stack
#pragma -p2
#pragma profile_statements
```

You can also specify pragmas from the compiler command-line using:

`-zpLetterDigit`

or

`-zp[no_]FeatureName`

Table 4-2 lists the pragmas recognized by the ARM C++ compilers. Values marked with an asterisk (*) are the default values.

Thumb The options marked with † are not available in Thumb

Table 4-2: Pragmas

	Pragma Name	Short Form		'no' form	
†	check_memory_accesses	c1		c0	*
	check_printf_formats	v1		v0	*
	check_scanf_formats	v2		v0	*
	check_stack	s0 [a]		s1 [b]	
	continue_after_hash_error	e1		e0	*
	force_top_level	t1		t0	*
†	FP register variable	f1-f4		f0	*
	include_only_once	i1		i0	*
	integer register variable	r1-r6		r0	*

	Pragma Name	Short Form		‘no’ form	
	optimise_crossjump	j1	*	j0	
	optimise_cse	z1	*	z0	
	optimise_multiple_loads	m1	*	m0	
†	profile	p1		p0	*
†	profile_statements	p2		p0	*
	side_effects	y0	*	y1	
	warn_deprecated	d1	*	d0	
	warn_implicit_fn_decls	a1	*	a0	

a. This is the default for ARM
b. This is the default for Thumb.

The following sections describe these pragmas in more detail.

Pragmas controlling the preprocessor

The following pragmas control aspects of the preprocessor. Refer to *Setting preprocessor options* on page 4-14 for descriptions of other preprocessor options that can be set from the command-line.

`continue_after_hash_error`

Compilation continues after `#error`.

`include_only_once`

The containing `#include` file is included only once. If its name recurs in a subsequent `#include` directive, the directive is ignored.

`force_top_level`

The containing `#include` file should only be included at the top level of a file. A syntax error results if the file is included within a function.

Pragmas controlling printf/scanf argument checking

The following pragmas control type checking of printf-like and scanf-like arguments.

`check_printf_formats`

Controls whether the actual arguments to printf-like functions are type-checked against the format designators in a literal format string. Calls using non-literal format strings cannot be checked. By default, all calls involving literal format strings are checked. For example:

```
#pragma check_printf_formats
extern void myprintf(const char *, format,...);//printf format
#pragma no_check_printf_formats
```

`check_scanf_formats`

Controls whether the actual arguments to scanf-like functions are type-checked against the format designators in a literal format string. Calls using non-literal format strings cannot be checked. By default, all calls involving literal format strings are checked. For example:

```
#pragma check_scanf_formats
extern void myformat(const char *, format,...);//scanf format
#pragma no_check_scanf_formats
```

Pragmas controlling optimization

The following pragmas allow fine control over where specific optimizations are applied. Note that correct use of the **volatile** qualifier should make this degree of control unnecessary. **volatile** is also available in the PCC mode unless `-strict` is specified. Refer to *Controlling optimization* on page 4-19 for information on optimization options that are specified from the compiler command-line.

`optimise_crossjump`

Controls cross-jumping (the “common tail” optimization). This is disabled if `-Otime` is specified.

`optimise_multiple_loads`

Controls the optimization of multiple load (LDR) instructions to a single load multiple (LDM) instruction.

`optimise_cse` Controls common sub-expression elimination.

`no_side_effects`

Asserts that all function declarations up to the next `#pragma side_effects` describe pure functions.

Functions that are pure are candidates for common sub-expression elimination. By default, functions are assumed to be impure. That is, it is assumed that they have side-effects. This pragma allows you to tell the compiler that specific functions are candidates for common sub-expression elimination.

Note that a function is only properly defined as pure if its result depends only on the value of its scalar argument. This means that they cannot use global variables, or dereference pointers, because the compiler assumes that the function does not access memory at all, except for the stack. When called twice with the same parameters, it must return the same value. See also the description of **__pure** in the section *Function declaration keywords* on page 4-32.

Pragmas controlling code generation

The following pragmas control how code is generated. Many other code generation options are available from the compiler command-line. Refer to *Controlling code generation* on page 4-19 for more information.

`no_check_stack`

If stack checking is enabled with `-apcs /swst`, this pragma disables the generation of code at function entry that checks for stack limit violation. This is possible only if the function uses less than 256 Kilobytes of stack.

Note that you must use `no_check_stack` if you are writing a signal handler for the SIGSTAK event. When this event occurs, stack overflow has already been detected. Checking for it again in the handler results in fatal, circular recursion.

Refer to Chapter 5 of the SDT 2.11 *Reference Guide* for information on the ARM Procedure Call Standard.

`check_memory_accesses`

This pragma instructs the compiler to precede each access to memory by a call to the appropriate one of:

```
__rt_rd?chk
__rt_wr?chk
```

where `?` equals 1, 2, or 4 for byte, **short**, and **long** writes respectively. It is up to your library implementation to check that the address given is reasonable.

Pragmas controlling global register variables

The ARM C++ compilers recognize a number of pragmas that can be used to assign global register variables. These pragmas have no long-form counterparts. The pragmas are:

`-r1 to -r6` Introduces a list of **extern**, file-scope, integral type register variable declarations. Global register variables cannot be qualified or initialized at declaration. Valid types are:

- Any integer type.
- Any pointer type.

`-r0` Terminates the list of global integral type register variables.

`-f1 to -f4` Introduces a list of **extern**, file-scope, floating-point register variable declarations. Any floating-point type can be allocated to a floating-point register. These pragmas are only applicable if a `-apcs /hardfp` is in use.

`-f0` Terminates the list of global floating-point register variables.

Each declaration declares a name for the same register variable. For example:

```
#pragma -r1                  // 1st global register
int *sp;
#pragma -r2                  // 2nd global register
int *fp, *ap;                // synonyms
#pragma -r0                  // end of global declaration
#pragma -f1
double pi;                   // 1st global FP register
#pragma -f0
```

Table 4-3 shows how the pragmas relate to APCS register names and machine registers.

Table 4-3: Global Register Variables

Pragma name	APCS register	Machine register
-r1	v1	R4
-r2	v2	R5
-r3	v3	R6
-r4	v4	R7
-r5	v5	R8
-r6	sb/v6	R9
-f1	f4	F4
-f2	f5	F5
-f3	f6	F6
-f4	f7	F7

Depending on the APCS variant in use, between five and seven integer registers and four floating-point registers are available for use as global register variables. In practice, using more than three global integer register variables and two global floating-point register variables is *not* recommended.

Unlike register variables declared with the standard C++ **register** keyword, the compiler will *not* move global register variables to memory as required. If you declare too many global variables, code size will increase significantly or, in some cases, your program may not compile.

Note the following important points:

- You must exercise care when using global register variables. There is no check at link time to ensure that direct calls are sensible. If possible, any global register variables used in a program should be defined in each compilation unit of the program. In general, it is best to place the definition in a global header file.
- Because a global register variable maps to a callee-saved register, its value is saved and restored across a call to a function in a compilation unit that does not use it as a global register variable, such as a library function.
- Calls back into a compilation unit that uses a global register variable are dangerous. For example, if a global register using function is called from a compilation unit that uses the register as a compiler allocated register, the function will probably read the wrong values from its supposed global register variables.

Refer to *Variable declaration keywords* on page 4-33 for information how to allocate global register variables using keywords.

4.4.2 Function declaration keywords

Several function declaration keywords tell the compiler to give a function special treatment. These function declaration keywords are not portable to other C++ compilers.

__inline This allows C functions to be inlined. The semantics of **__inline** are exactly the same as the C++ **inline** keyword:

```
__inline int f(int x) {return x*5+1:}
int f(int x, int y) {return f(x), f(y);}
```

The compiler always inlines functions when **__inline** is used. Code density and performance could be adversely affected if large functions are inlined.

__irq This allows a C++ function to be used as an interrupt routine. All registers, except floating-point registers, are preserved, not just those normally preserved under the APCS. Also, the function is exited by setting the PC to lr–4 and the PSR to its original value. This is not available in tcpp.

__pure Asserts that a function declaration is pure.

Functions that are pure are candidates for common sub-expression elimination. By default, functions are assumed to be impure. That is, it is assumed that they have side-effects. This keyword allows you to tell the compiler that specific functions are candidates for common sub-expression elimination. **__pure** has the same effect as the `no_side_effects` pragma.

Note that a function is only properly defined as pure if its result depends only on the value of its scalar argument. This means that they cannot use global variables, or dereference pointers, because the compiler assumes that the function does not access memory at all, except for the stack. When called twice with the same parameters, it must return the same value.

__swi A SWI taking up to four integer-like arguments in registers R0 to R(*argcount* -1), and returning up to four results in registers R0 to R(*resultcount*-1), can be described by a C++ function declaration. This causes function invocations to be compiled inline as a SWI.

For a SWI returning 0 results use:

```
void __swi(swi_number) swi_name(int arg1, ..., int argn);
```

For example:

```
void __swi(42) terminate_process(int arg1, ..., int argn);
```

For a SWI returning 1 result, use:

```
int __swi(swi_number) swi_name(int arg1, ..., int argn);
```

For a SWI returning more than 1 result use:

```
struct { int res1, ... resn }
__value_in_regs
__swi(swi_number) swi_name(int arg1, ... int argn);
```

Note that **__value_in_regs** is needed to specify that a small non-packed structure of up to four words (16 bytes) is returned in registers, rather than by the usual structure passing mechanism specified in the ARM Procedure Call Standard.

__swi_indirect An indirect SWI that takes the number of the SWI to call as an argument in R12 can be described by a C++ function declaration such as:

```
int __swi_indirect(swi_indirect_number)
swi_name(int real_swi_number, int arg1, ... argn);
```

where:

swi_indirect_number

is the SWI number used in the SWI instruction

real_swi_number

is the SWI number used in the SWI instruction and may be specified at function call.

For example:

```
int __swi_indirect(0) ioctl(int swino, int fn, void *argp);
```

This might be called as:

```
ioctl(IOCTL+4, RESET, NULL);
```

It compiles to a SWI 0 with IOCTL+4 in R12.

Note that your system must support **__swi_indirect**.

__value_in_regs

This allows the compiler to return a structure in registers rather than using the stack. For example:

```
typedef struct int64_struct {
unsigned int lo;
unsigned int hi;
} int64;
__value_in_regs extern int64 mul64(unsigned a, unsigned b);
```

See Chapter 5 of the SDT 2.11 *User Guide* for information on the default method of passing and returning structures.

4.4.3 Variable declaration keywords

This section describes the implementation of various standard C++ and ARM-specific variable declaration keywords. Standard C++ keywords that do not have ARM-specific behavior or restrictions are not documented.

Standard keywords

register You can declare any number of auto variables to have the storage class **register**. Depending on the variant of the ARM Procedure Call Standard (APCS) that is in use, there are between five and seven integer registers available, and four floating-point registers.

In general, declaring more than four integer register variables and two floating-point register variables is not recommended.

Objects of the following types can be declared to have the **register** storage class:

- Any integer type.
- Any pointer type.
- Any integer-like structure, such as any one word **struct** or **union** in which all addressable fields have the same address, or any one word structure containing bitfields only. The structure must be padded to 32 bits.
- A floating-point type, if software floating-point is used. The double precision floating-point type **double** occupies two ARM registers if software floating-point emulation is used.

Note that the **register** keyword is regarded by the compiler as a suggestion only. Other variables, not declared with the **register** keyword, may be held in registers for extended periods and register variables may be held in memory for some periods.

ARM-specific keywords

The following variable declaration keywords allow you to specify that a declared variable is allocated to a global register variable. These keywords work in the same way as the `#pragma -rn` and `#pragma -fn` directives. Note that the global register, whether specified by keywords or by pragmas, must be specified in all declarations of the same variable. For example, the following is an error:

```
int x;
__global_reg(1) x;
```

In addition, **__global_reg** variables cannot be initialized at definition. For example, the following is an error:

```
__global_reg(1) int x=1;              // error
```

Refer to the section *Pragmas controlling global register variables* on page 4-30 for a more complete description of global register variables.

The following variable declaration keywords are available:

__global_reg(n)

Allocates the declared variable to a global integer register variable, in the same way as `#pragma -rn`. The variable must have an integral or pointer type.

__global_freg(n)

Allocates the declared variable to a global floating-point register variable, in the same way as `#pragma -fn`. The variable must have type **float** or **double**. This keyword applies only if `-apcs /hardfp` is in use.

4.4.4 Type qualifiers

This section describes the implementation of various standard C++ and ARM-specific type qualifiers. These type qualifiers can be used to instruct the compiler to treat the qualified type in a special way. Note that **__packed** is not, strictly speaking, a type qualifier. It is included in this section because it behaves like a type qualifier in most respects. Standard C++ qualifiers that do not have ARM-specific behavior or restrictions are not documented.

Access to packed data is very expensive on the ARM processor. In general you should minimize data accesses through packed structures to avoid performance loss. Refer to *Structured data types* on page 6-10 for a detailed description of how ARM C++ implements structure packing.

__packed There is no command-line option to change the default packing of structures. Packed structures must be specified with the type qualifier **__packed**.

If you wish to use **packed** rather than **__packed**, you must define it:

```
#define packed __packed
```

__packed behaves as a type qualifier, like **volatile**, and may qualify any non-floating-point type. Floating-point types may not be fields of packed structures.

There is no difference between `int x` and `__packed int x`. However, there is a significant difference between `int *px` and `__packed int *px` when `px` is de-referenced. In the latter case, an **int** will be correctly loaded from a location of unknown alignment.

A **__packed struct** or **__packed union** type must be declared explicitly. It is a different type from the corresponding non-packed type and its packedness is an attribute of its **struct** or **union** tag. This means that **__packed** is more than just a type qualifier. Any variables declared using a **packed** tag automatically inherit the packed attribute, so **__packed** does not have to be specified:

```
__packed struct P { ... };
struct P pp;                          // pp is a packed struct
```

As a result, the following will be faulted:

```
struct Foo { ... };
__packed struct Foo PackedFoo;              // illegal
```

or

```
struct Foo { ... };
typedef __packed struct Foo PackedFoo;      // illegal
```

This ensures that a **__packed struct** can never be assignment-compatible with an unpacked **struct**. This would not be the case if **__packed** were a simple type qualifier like **volatile** and **const**. It is impossible to assign a packed structure to a non-packed structure except by using field by field copy.

Each field of a packed **struct** or packed **union** inherits the **__packed** qualifier.

There are no packed array types. A packed array is simply an array of objects of packed type. There is no inter-element padding.

The effect of casting away **__packed** is undefined. For example:

```
int f(__packed int *px)
{
return *(int *)px;            // undefined behavior
}
```

All top-level objects (global or local) are word-aligned and occupy an integral number of words of store unless `-zat` is set to less than 4 (the default for tcpp), so there may be padding between separately declared top-level packed structures.

volatile

The standard C++ qualifier **volatile** warns the compiler that the qualified type contains data that may be changed from outside the program. The compiler will not attempt to optimize accesses to **volatile** qualified types. For example, volatile structures can be mapped onto memory-mapped hardware.

In ARM C++ a **volatile** qualified object is *accessed* if any word or byte of the object is read or written. For **volatile** qualified objects, reads and writes occur as directly implied by the source code, in the order implied by the source code. The effect of accessing a **volatile short** is undefined for architecture 3 and lower.

Chapter 5

Mixed Language Programming

This chapter describes how to write mixed C, C++, and Assembler code. It contains the following sections:

5.1 Using C header files

This section describes how to use C header files from within your C++ code.

5.1.1 Including system C header files

To include standard system C header files, such as `stdio.h`, you need do nothing special. The standard C header files already contain the appropriate **extern** "C" directives. For example:

```
// C++ code

#include <stdio.h>
int main()
{
        //...
        return 0;
}
```

The C++ standard specifies that the functionality of the C header files is available through C++ specific header files. These files are installed in `/include` and may be referenced in the usual way. For example:

```
// C++ code

#include <cstdio>
int main()
        // ...
        return 0;
}
```

In ARM C++, these headers simply `#include` the C headers.

5.1.2 Including your own C header files

To include your own C header files, you must wrap the `#include` directive in an **extern** "C" statement. You can do this in two ways:

- When the file is `#included`. For example:

  ```
  // C++ code

  extern "C"{
  #include "my-header1.h"
  #include "my-header2.h"
  }

  int main()
  {
    // ...
    return 0;
  }
  ```

- By adding the **extern** "C" statement to the header file. For example:

  ```
  /* C header file */

  #ifdef __cplusplus       /* Insert the start of the extern C construct */
  extern "C" {
  #endif
  ```

```
/* Body of header file */

#ifdef __cplusplus       /* Insert the end of the extern C construct */
}
#endif
```

The C header file can now be `#included` in either C or C++ code.

5.2 Using Inline Assembler

The ARM inline assembler allows you to use ARM Assembler instructions within a C++ program. You can use inline Assembler to:

- achieve more efficient code
- use features of the target processor that the compiler cannot use.

Inline Assembler is most useful in cases where a relatively small amount of Assembler code is needed.

The inline assembler supports very flexible interworking with C++. Any register operand may be an arbitrary C++ expression. The inline assembler also auto expands complex instructions and optimizes the Assembler code.

The armcpp inline assembler implements the full ARM instruction set, including generic coprocessor instructions, halfword instructions and long multiply. The tcpp inline assembler implements the full Thumb instruction set.

5.2.1 Invoking the inline assembler

The ARM C++ compilers handle the syntax proposed in the Draft Standard, with the restriction that the string-literal mustAssembler be a single string. For example:

```
asm("instruction[;instruction]");
```

The **asm** declaration must be inside a C or C++ function. You cannot include comments in the string literal.

In addition to the syntax proposed in the Draft Standard, ARM C++ supports the following syntax.

An inline Assembler block starts with the assembler specifier **asm**, followed by one or more Assembler instructions inside braces:

```
asm
{
instruction [; instruction]                 // Comment
[instruction]
}
```

If two instructions are on the same line, a semicolon must be used as a separator. A single instruction cannot be spread over multiple lines unless the separator is used. C or C++ comments may be used anywhere within an inline Assembler block.

An **asm** statement may be used anywhere a C++ statement is expected. The **__asm** keyword is a synonym supported for compatibility with C.

5.2.2 Assembler instruction set

The ARM and Thumb instruction sets are described in the *ARM Architecture Reference Manual* (ARM DDI-0100). All instruction opcodes and register specifiers may be written in either lowercase or uppercase.

Operand expressions

Any register or constant operand may be an arbitrary C++ expression, so that C++ variables may be read or written. The compiler may add code to evaluate C++ expressions and allocate them to registers. When an operand is used as a destination, the expression must be assignable (an lvalue).

When writing code which uses both physical registers and C++ expressions you must take care not to use complex expressions that require too many registers to evaluate. The compiler will issue an error message if it detects conflicts during register allocation.

Constants

The constant expression specifier '#' is optional. If it is used, the expression following must be constant.

Note *The notation to specify the actual rotate of a 8-bit constant is not available in inline Assembler. This means that where an 8-bit shifted constant is used, the C flag should be regarded as corrupted if the PSR is updated.*

Instruction expansion

The range of constants is not limited to the default range of the instruction. All ARM and Thumb instructions with a constant operand support instruction expansion. In addition, the MUL instruction can be expanded into a sequence of adds and shifts when the third operand is a constant.

The effect of updating the PSR by an expanded instruction is:

- Arithmetic instructions set the NZCV flags correctly
- Logical instructions:
  - set the NZ flags correctly
  - do not change the V flag
  - corrupt the C flag.
- TEQP, TSTP, and MRS set the NZCV flags correctly

Labels

C++ labels are allowed. Labels can be used by branch instructions only in the form:

```
B<cond> <label>
```

Storage declarations

All storage can be declared in the host program and passed to the inline assembler using C++ variables. Therefore, the storage declarations that are permitted in ordinary Assembler code are not implemented.

SWI/BL

SWIs and branches must specify exactly which calling standard is used. After the normal instruction fields, three optional register lists follow to specify:

- the registers that are the input parameters
- the registers that are output parameters after return
- the registers that are updatable by the called function.

```
SWI<cond> <number>, {<input_regs>}, {<output_regs>}, {<corrupted_regs>}
BL<cond> <function>, {<input_regs>}, {<output_regs>}, {<corrupted_regs>}
```

An omitted list is assumed to be empty, except for BL, which always corrupts LR, R0-R3, and IP.

The register lists have the same syntax as LDM/STM register lists. The condition code register can be specified as PSR.

5.2.3 Differences between inline assembler and armasm/tasm

There are a number of differences between Assembler accepted by the C++ inline assembler and Assembler accepted by the ARM and Thumb assemblers. For the inline assembler:

- The `'.'` notation as a shorthand for PC is not supported.
- The `LDR Rn, =expression` notation is not supported. Use `MOV Rn, expression` instead.
- Label expressions are not supported.
- ADRL is not supported.

5.2.4 Restrictions

The following restrictions apply to the use of inline Assembler:

- It is illegal to write to:
  - PC and SP
  - FP, SL, and SB, depending on the selected calling standard.

  Other registers, such as IP, LR, R0-R3, and PSR must be used with caution. If you use C or C++ expressions, these may be used as temporary registers by the compiler when evaluating the expression.
- LDM and STM instructions only allow physical registers to be specified in the register list.
- BX is not implemented.
- You can change processor modes, and alter the state of FP coprocessors, but the compiler is unaware of the change. If you change to a different processor mode, you must not use C expressions or C++ expressions until you change back to the original mode. Similarly, if you change the state of a FP coprocessor by executing FP instructions, you must not use floating point expressions until the original state has been restored.

5.2.5 Usage

The following points apply to using inline Assembler:

- Expressions with the comma operator must be bracketed.

  ```
  asm("ADD x, y, (f(), z)");
  ```
- The `&` operator cannot be used to denote hexadecimal constants. Use the '`0x`' prefix instead. For example:

  ```
  adm("AND x, y, 0xF00");
  ```
- If you are using physical registers, you must ensure that the compiler does not corrupt them when evaluating expressions. For example:

  ```
  asm
  {
    MOV R0, x
    ADD y, R0, x / y              // (x / y) overwrites R0 with the result
  }
  ```

  Because the compiler uses a function call to evaluate `x / y`, it corrupts R2, R3, IP, LR, PSR and alters R0 and R1. The value in R0 is lost. The compiler can detect the corruption in many cases, for example when it needs a temporary register and the register is already in use:

```
asm
{
  MOV ip, #3
  ADD x, x, #0x12345678         // this instruction is expanded
  ORR x, x, ip
}
```

The compiler uses IP as a temporary register while expanding the add-instruction, corrupting the value 3 in IP. An error is issued.

- Do not use physical registers to address variables, even when it seems obvious that a specific variable is mapped onto a specific register. If the compiler detects this it either generates an error or puts the variable into another register to avoid conflicts:

```
int bad_f(int x)                // x in R0
{
  asm
        {
                ADD R0, R0, #1 // wrongly asserts that x is still in R0
        }
  return x; // x in R0
}
```

This code returns `x` unaltered. The compiler assumes that `x` and R0 are two different variables, despite the fact that `x` is allocated to R0 on both function entry and function exit. As the Assembler code does not do anything useful, it is optimized away.

5.2.6 Examples

Example 1: String copying

```
void my_strcpy(char *src, char *dst)
{
       int ch;
       asm
       {
              loop:
                     LDRB ch, [src], #1
                     STRB ch, [dst], #1
                     CMP ch, #0
                     BNE loop
       }
}
```

Example 2: Function call

```
#include <stream.h>
int int_sum(int a, int b)            // uses only R0 and R1
{                                    // (unless compiled for debug!)
       return a + b;                 // returns R0
}
int sum_array(int arr[], int n)
{
       int sum = 0;
       while (--n >= 0)
       asm
       {
              MOV  R0, sum
              MOV  R1, arr[n]
              BL   int_sum__FiT1, {R0, R1}, {R0}, {R1, R2, R3, LR}
                                     // note name mangling
              MOV  sum, R0
       }
       return sum;
}

int main(void)
{
       int x[] = { 1, 3, 5, 7, 100 };
       int_sum(1, 2);
       cout << sum_array(x, 5) << '\n';
       return 0;
}
```

5.3 Calling between C, C++, and Assembler

This section provides examples that may help you to call C and Assembler code from C++, and to call C++ code from C and Assembler. It also describes calling conventions and data types.

Note *The information in this section is implementation-dependent and may change in future versions ARM C++.*

5.3.1 Calling Conventions

ARM C++ uses the same calling conventions as ARM C with the following exceptions:

- When an object of **struct** or **class** type is passed to a function and the type has an explicit copy constructor, the object is passed by reference and the called function makes a copy.
- Non-static member functions are called with the implicit ‘this’ parameter as the first argument, or as the second argument if the called function returns a non-int-like **struct**.

5.3.2 Data types

ARM C++ uses the same data types as ARM C with the following exceptions and additions:

- C++ objects of type **struct** or **class** have the same layout as would be expected from the ARM C compiler if and only if they have no base classes or virtual functions.
- References are represented as pointers.

  Pointers to data members and pointers to member functions occupy four bytes. They have the same null pointer representation as normal pointers.
- No distinction is made between pointers to C functions and pointers to C++ (non-member) functions.

5.3.3 Symbol name mangling

ARM C++ mangles external names of functions and static data members in a manner similar to that described in section 7.2c of Ellis, M.A. and Stroustrup, B., *The Annotated C++ Reference Manual* (1990). The linker and decaof unmangle symbols.

C names must be declared as **extern** "C" from within C++ programs. This is done already for the ARM ANSI C headers. Refer to *Using C header files* on page 5-2 for more information.

5.3.4 Examples

The following code examples demonstrate how to:

- call C and Assembler functions from within C++
- call C++ functions from C and Assembler
- call a non-static, non-virtual C++ member function from C or Assembler
- pass references to and from a C function.

Note that you should *not* rely on the following implementation details:

- name mangling
- passing the implicit 'this' parameter
- calling virtual functions
- representation of references
- layout of C++ class types that are not also C types
- passing of class objects that have copy constructors.

These are subject to change in future releases of ARM C++.

The following general rules apply to mixed language programming:

- Use C calling conventions.
- In C++, non-member functions may be declared as **extern** "C" to specify that they have C linkage. In this release of ARM C++, having C linkage means that the symbol defining the function is not mangled. C linkage can be used to implement a function in one language and call it from another. Note that functions that are declared **extern** "C" cannot be overloaded.

The following rules apply to calling C++ functions from C and Assembler:

- To call a global (non-member) C++ function declare it **extern** "C" to give it C linkage.
- Member functions (both static and non-static) always have mangled symbols. You can determine the mangled symbol by using `decaof -sm` on the object file defining the function.
- Do not try to access C++ objects or structures that do not also exist in C.
- C++ inline functions cannot be called from C unless special arrangements are made to ensure that the C++ compiler generates an out-of-line copy of the function. For example, taking the function address will result in an out-of-line copy.
- Non-static member functions receive the implicit 'this' parameter as a first argument in R0, or as a second argument in R1 if the function returns a non-int-like structure. Static member functions do not receive an implicit 'this' parameter.

Example 1: Calling a C function from C++

In C++:

```
struct S {                    // has no base classes nor virtual functions
       S(int s) : i(s) { }
       int i;
};
extern "C" void cfunc(S *);   // the declaration of the C function to
                              // be called from C++

int f() {
       S s(2);                // initialise 's'
       cfunc(&s);             // call 'cfunc' so that it can change 's'
       return s.i * 3;
}
```

In C:

```
struct S {
       int i;
};

void cfunc(struct S *p) {     /* the definition of the C function to be called
                              from C++ */
       p->i += 5;
}
```

Example 2: Calling Assembler from C++

In C++:

```
struct S {                    // has no base classes nor virtual functions
       S(int s) : i(s) { }
       int i;
};

extern "C" void asmfunc(S *); // the delcaration of the Asm function
                              // to be called

int f() {
       S s(2);                // initialise 's'
       asmfunc(&s);           // call 'asmfunc' so that it can change 's'
       return s.i * 3;
}
```

In Assembler:

```
       AREA Asm, CODE
       EXPORT asmfunc
asmfunc               ; the definition of the Asm function to be called from C++
       LDR r1, [r0]
       ADD r1, r1, #5
       STR r1, [r0]
       MOV pc, lr
       END
```

Example 3: Calling C++ from C

In C++:

```
struct S {                    // has no base classes nor virtual functions
       S(int s) : i(s) { }
       int i;
};

extern "C" void cppfunc(S *p) {       // the definition of the C++ function
                                      // to be called from C
                                      // the function is still written in C++,
                                      // only the linkage is C
       p->i += 5;
}
```

In C:

```
struct S {
       int i;
};

extern void cppfunc(struct S *p);     /* the declaration of the C++
                                      function to be called from C */

int f() {
       struct S s;
       s.i = 2;                       /* initialise 's' */
       cppfunc(&s);           /* call 'cppfunc' so that it can change 's' */
       return s.i * 3;
}
```

Example 4: Calling a C++ function from Assembler

In C++:

```
struct S {                    // has no base classes nor virtual functions
       S(int s) : i(s) { }
       int i;
};
extern "C" void cppfunc(S * p) {      // the definition of the C++
                                      // function to be called from Asm
                                      // the body is C++, only the linkage is C
     p->i += 5;
}
```

In ARM Assembler:

```
       AREA Asm, CODE
       IMPORT cppfunc         ; import the name of the C++ function to
                              ; be called from Asm

       EXPORT f
f
       STMDB sp!,{lr}
       MOV r0,#2
       STR r0,[sp,#-4]!       ; initialise struct
       MOV r0,sp              ; argument is pointer to struct
       BL cppfunc             ; call 'cppfunc' so that it can change the struct
       LDR r0, [sp], #4
       ADD r0, r0, r0,LSL #1
       LDMIA sp!,{pc}
       END
```

Example 5: Calling a C++ member function from C or Assembler

The following code demonstrates how to call a non-static, non-virtual C++ member function from C or Assembler.

In C++:

```
struct T {
       T(int i) : t(i) { }
       int t;
       int f(int i);
};

int T::f(int i) { return i + t; }     // definition of the C++ function
                                      // to be called from C

extern "C" int cfunc(T*);             // declaration of the C function to be
                                      // called from C++

int f() {
       T t(5);                        // create an object of type T
       return cfunc(&t);
}
```

In C:

```
struct T;
extern int f__1TFi(struct T*, int);   /* the mangled name of the C++
                                      function to be called */

int cfunc(struct T* t) {              /* definition of the C function to be
                                       called from C++ */
       return 3 * f__1TFi(t, 2);      /* like '3 * t->f(2)' */
}
```

Or, implementing `cfunc` in ARM Assembler:

```
       EXPORT cfunc
       AREA cfunc, CODE
       IMPORT f__1TFi

       STMDB sp!,{lr}                 ; r0 already contains the object pointer
       MOV r1, #2
       BL f__1TFi
       ADD r0, r0, r0, LSL #1         ; multiply by 3
       LDMIA sp!,{pc}
       END
```

Example 6: Passing a reference to and from a C function

In C++:

```
extern "C" int cfunc(const int&);     // declaration of the C function
                                      // to be called from C++

extern "C" int cppfunc(const int& r) {       // definition of the C++ to be
                                             // called from C
        return 7 * r;
}

int f() {
        int i = 3;
        return cfunc(i);              // passes a pointer to 'i'
}
```

In C:

```
extern int cppfunc(const int*);       /* declaration of the C++ to be called
                                      from C */

int cfunc(const int* p) {             /* definition of the C function to
                                      be called from C++ */
        int k = *p + 4;
        return cppfunc(&k);
}
```

Chapter 6

ARM C++ Compiler Reference

This chapter describes the reference information you may need in order to make effective use of ARM C++. It contains the following sections:

6.1 C++ Language Feature Support

The ARM C++ compilers support the majority of the language features described in the ISO/IEC December 1996 Draft Standard for C++. This section lists the C++ language features defined in the Draft Standard, and states whether or not that language feature is supported by ARM C++.

6.1.1 Major language feature support

Table 6-1 shows the major language features supported by this release of ARM C++.

Table 6-1: Major language feature support

Major Language Feature	Draft Standard Section	Supported
Core language	1 to 13	Yes
Templates	14	Partial. Templates are supported except for the export feature.
Exceptions	15	Partial
Libraries	17 to 27	Partial. Refer to *The Standard C++ Library* on page 6-4.

6.1.2 Minor language feature support

Table 6-2 shows the minor language features supported by this release of ARM C++.

Table 6-2: Minor language feature support

Minor Language Feature	Supported
Namespaces	No
Runtime type identification (RTTI)	Partial. **Typeid** is supported for static types and expressions with non-polymorphic type. See also the restrictions on new style casts.
New style casts	Partial. ARM C++ supports the syntax of new style casts, but does not enforce the restrictions. New style casts behave in the same manner as old style casts.
Array new/delete	No
Nothrow **new**	No, but **new** does not throw.
bool type	Yes
wchar_t type	No
explicit keyword	No
Static member constants	No
extern inline	Partial. This is supported except for functions that have static data.
Full linkage specification	Yes
for loop variable scope change	Yes
Covariant return types	No
Default template arguments	Yes
Template instantiation directive	Yes
Template specialization directive	Yes
typename keyword	Yes
Member templates	Yes
Partial specialization for class template	Yes
Partial ordering of function templates	Yes
Universal character names	No

6.2 The Standard C++ Library

This section describes the Rogue Wave Standard C++ library that is supplied with ARM C++. Version 1.2.1 of the Rogue Wave library provides a subset of the library defined in the January 1996 Draft Standard. There are slight differences to the December 1996 version of the Draft Standard.

The Standard C++ library is distributed in binary form only. It is built into the armcpplib library, together with additional library functions described in *Using the ARM C++ library* on page 4-3. The library is also supplied as pre-built object files to enable you to rebuild armcpplib if required. Refer to *Rebuilding the ARM C++ Library* on page 6-5 for more information on rebuilding armcpplib.

Table 6-3 lists the library features that are supported in version 1.2.1 of the library. The most significant features missing from this release are:

- **iostream**
- **locale**
- **valarray**
- **typeinfo**.

Note that **iostream** and **typeinfo** are supported in a basic way by the ARM C++ library additions. Refer to *Using the ARM C++ library* on page 4-3 for more information.

For detailed information on the Rogue Wave Standard C++ library, refer to the Rogue Wave HTML documentation that is included with this release of ARM C++.

Table 6-3: Standard C++ library support

Draft Standard Section	Library Feature
18.2.1	Numeric limits
19.1	Exception classes
20.3	Function objects
20.4.2	Raw storage iterator
20.4.3	Memory handling primitives
20.4.4	Specialized algorithms for raw storage
20.4.5	Template class **auto_ptr**
21	Strings library
23	Containers library
24	Iterators library (except sections 24.4.3 and 24.4.4)
25	Algorithms library
26.2	Complex number
26.4	Generalized numeric operations

6.3 Rebuilding the ARM C++ Library

The ARM C++ library consists of two subsections:

- Prebuilt sublibrary files for the Rogue Wave Standard C++ library. These are installed in your SDT 2.11 `/rebuild/stdlib` directory.
- Source code and rebuild scripts for the ARM C++ library additions.

The source for the ARM C++ library additions may be modified as you wish. The source for the Rogue Wave Standard C++ library is not freely distributable. It may be obtained from Rogue Wave Software Inc., or through ARM Ltd, for an additional licence fee.

Follow these step to rebuild a particular variant of the armcpplib library, for example `armcpplib_hc.32l`:

1. Ensure that armcpp (and/or tcpp, if building 16-bit libraries) and armlink are in your path.
2. Change directory to `/rebuild/cpplib/cpplib.b/`*`platform`* where *`platform`* is one of `cchppa, gccsolrs, gccsunos, intelrel, alpharel`.
3. Enter `make lib_cpplib_hc.32l`. On the PC, enter `nmake`.

   This creates `cpplib_hc.32l`. Some compilation warnings are normal.
4. Create a temporary directory and copy the new `cpplib_c.32l` to it.
5. Copy `/rebuild/stdlib/stdlib_hc.32l` to your temporary directory.
6. Use these three commands to extract the object files from the two libraries and create a new armcpplib library:

   ```
   armlib -e cpplib_hc.32l \*
   armlib -e stdlib_hc.32l \*
   armlib -c -o armcpplib_hc.32l *.o*
   ```
7. The new library is available for use and the temporary directory may be deleted.

6.4 C++ Implementation Details

This section describes implementation details for the ARM C++ compilers, including:

- character sets and identifiers
- sizes, ranges and implementation details for basic data types
- implementation details for structured data types
- implementation details for bitfields.

6.4.1 Character sets and identifiers

The following points apply to the character sets and identifiers expected by the compiler:

- An identifier can be of any length. The compiler truncates an identifier after 256 characters, all of which are significant.
- Uppercase and lowercase characters are distinct in all internal and external identifiers. In PCC mode and limited PCC mode, an identifier may also contain a dollar `($)` character.
- Calling `setlocale(LC_CTYPE, "ISO8859-1")` makes the `isupper()` and `islower()` functions behave as expected over the full 8-bit Latin-1 alphabet, rather than over the 7-bit ASCII subset.
- The characters in the source character set are assumed to be ISO 8859-1 (Latin-1 Alphabet), a superset of the ASCII character set. The printable characters are those in the range 32 to 126 and 160 to 255. any printable character may appear in a string or character constant, and in a comment.
- ARM C++ has no support for multi-byte character sets.
- Other properties of the source character set are host specific.

The properties of the execution character set are target-specific. The ARM C++ library supports the ISO 8859-1 (Latin-1 Alphabet) character set, so the following points are valid:

- The execution character set is identical to the source character set.
- There are eight bits in a character in the execution character set.
- There are four **char**s/bytes in an **int**. If the memory system is:

  little-endian — the bytes are ordered from least significant at the lowest address to most significant at the highest address

  big-endian — the bytes are ordered from least significant at the highest address to most significant at the lowest address
- A character constant containing more than one character has the type **int**. Up to four characters of the constant are represented in the integer value. The first character in the constant occupies the lowest-addressed byte of the integer value. Up to three following characters are placed at ascending addresses. Unused bytes are filled with the NULL (“\0”) character.
- All integer character constants that contain a single character or character escape sequence are represented in both the source and execution character sets (by an assumption that may be violated in any given retargeting of the generic ARM C library).
- Characters of the source character set in string literals and character constants map identically into the execution character set (by an assumption that may be violated in any given retargeting of the generic ARM C library).
- No locale is used to convert multi-byte characters into the corresponding wide characters (codes) for a wide character constant (not relevant to the generic implementation).

6.4.2 Basic data types

This section provides information about how the basic data types are implemented in ARM C++.

Size and alignment of basic data types

Table 6-4 gives the size and natural alignment of the basic data types. Note that type alignment varies, depending on the context in which the type is used:

- The alignment of top level static objects such as global variables is the maximum of the natural alignment for the type, and the value set by the `-zat` compiler option. For example, if `-zat1` is specified a global **char** variable has an alignment of 1. This option is described in *Alignment options* on page 4-21.
- Local variables are always word aligned. For example, a local **char** variable has an alignment of 4.
- Fields in unpacked structured data types are always aligned to the natural alignment for the type. Fields in packed structured data types have an alignment of 1.

Table 6-4: Size and alignment of data types

Type	Size in bits	Natural alignment
char	8	1 (byte aligned)
short	16	2 (halfword aligned)
int	32	4 (word aligned)
long	32	4 (word aligned)
long long	64	4 (word aligned)
float	32	4 (word aligned)
double	64	4 (word aligned)
long double	64 (subject to change)	4 (word aligned)
all pointers	32	4 (word aligned)
bool	32	4 (word aligned)

Char

The following points apply to the **char** data type:

- Data items of type **char** are unsigned by default. In C++ mode and ANSI C mode they may be explicitly declared as **signed char** or **unsigned char**.
- In PCC mode there is no **signed** keyword. Therefore a **char** is signed by default and may be declared unsigned if required.

Integer

Integers are represented in two's complement form. Refer to *Operations on integral types* on page 6-8 for more information on integers.

Float

Floating-point quantities are stored in IEEE format. In **double** and **long double** quantities, the word containing the sign, the exponent, and the most significant part of the mantissa is stored at the lower machine address.

Pointers

The following remarks apply to pointer types other than pointers to members:

- Adjacent bytes have addresses that differ by one.
- The macro `NULL` expands to the value 0.
- Casting between integers and pointers results in no change of representation.
- The compiler warns of casts between pointers to functions and pointers to data, except when PCC mode is specified.
- The type **size_t** is defined as **unsigned int**, except in PCC mode where it is signed.
- The type **ptrdiff_t** is defined as **signed int**.

6.4.3 Operations on basic data types

The ARM C++ compilers perform the usual arithmetic conversions set out in section 5 of the Draft Standard. The following sections document additional points that should be noted with respect to arithmetic operations.

Operations on integral types

The following points apply to operations on the integral types:

- All signed integer arithmetic uses a two's complement representation.
- Bitwise operations on signed integral types follow the rules that arise naturally from two's complement representation. No sign extension takes place.
- Right shifts on signed quantities are arithmetic.
- Any quantity that specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from a shift of an unsigned or positive signed value. They yield -1 from a shift of a negative signed value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained as if by masking the original value to the length of the destination, and then sign extending the resulting integer. The same applies when an unsigned integer is converted to a signed integer of equal length.
- A conversion between integral types does not raise a processor exception.
- Integer overflow does not raise a processor exception.
- Integer division by zero raises a SIGFPE exception.

Operations on floating-point types

The following points apply to operations on floating-point types:

- Normal IEEE 754 rules apply.
- Rounding is to the nearest representable value by default.
- Conversion from a floating-point type to an integral type causes a floating-point exception to be raised only if the value cannot be represented in a **long int**, or **unsigned long int** in the case of conversion to an **unsigned int**.
- Floating-point underflow is not supported if software floating-point emulation is used. It is disabled by default for hardware floating-point coprocessors.
- Floating-point overflow raises a floating-point exception.
- Floating-point divide by zero raises a floating-point exception.

Pointer subtraction

The following remarks apply to pointers other than pointers to members:

- When two pointers are subtracted, the difference is obtained as if by the expression:

  ```
  ((int)a - (int)b) / (int)sizeof(type pointed to)
  ```

- If the pointers point to objects whose size is no greater than four bytes, the alignment of data ensures that the division will be exact, provided the objects are not packed.
- For longer types, such as **double** and **struct**, the division may not be exact unless both pointers are to elements of the same array. Also, the quotient may be rounded up or down at different times. This can lead to inconsistencies.

Expression evaluation

The compiler performs the usual arithmetic conversions (promotions) set out in the Draft Standard before evaluating an expression. The following should be noted:

- The compiler may re-order expressions involving only associative and commutative operators of equal precedence, even in the presence of parentheses. For example, a + (b - c) may be evaluated as (a + b) - c.
- Between sequence points, the compiler may evaluate expressions in any order, regardless of parentheses. Thus the side effects of expressions between sequence points may occur in any order.
- Similarly, the compiler may evaluate function arguments in any order.

Any detail of order of evaluation not prescribed by the Draft Standard may vary between releases of the ARM C++ compilers.

6.4.4 Structured data types

This section describes the implementation of the structured data types **union**, **enum**, and **struct**. It also discusses structure padding.

Unions

When a member of a **union** is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.

Enumerations

An object of type **enum** is implemented in the smallest integral type that contains the range of the **enum**. The type of an **enum** will be one of the following, according to the range of the **enum**:

- **unsigned char**
- **signed char**
- **unsigned short**
- **signed short**
- **signed int**.

Implementing **enum** in this way can reduce the size of the data area. The command-line option `-fy` sets the underlying type of **enum** to **signed int**. Refer to *Setting the source language* on page 4-11 for more information on the `-fy` option.

Non-packed structures

The following points apply to structures other than structures and classes with virtual functions or base classes:

Structure Alignment

The alignment of a non-packed structure is the larger of:

- The maximum alignment required by any of its fields.
- The minimum alignment for all structures, as set by the `-zas` compiler option. If the natural alignment of a structure is smaller than this, padding is added to the end of the structure. This option is described in *Alignment options* on page 4-21.

Field alignment

Structures are arranged with the first-named component at the lowest address. Fields are aligned as follows:

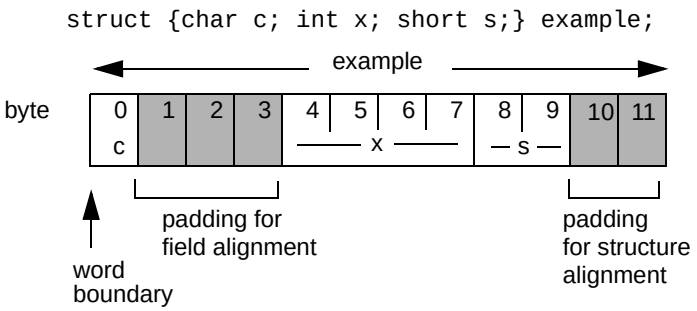
- A field with a **char** type is aligned to the next available byte.
- A field with a **short** type is aligned to the next even-addressed byte.
- Bitfield alignment depends on how the bitfield is declared. Refer to *Bitfields* on page 6-12 for more information.
- All other types are aligned on word boundaries.

Structures may contain padding to ensure that fields are correctly aligned, and that the structure itself is correctly aligned. For example, Figure 6-1 shows an example of a conventional, non-packed structure. In the example, bytes 1, 2, and 3 are padded to ensure correct field alignment. Bytes 10 and 11 are padded to ensure correct structure alignment.

The compiler pads structures in two ways, depending on how the structure is defined:

- Structures that are defined as **static** or **extern** are padded with zeroes.
- Structures that are defined with malloc or auto are padded with garbage. That is, pad bits will contain whatever was previously stored in memory. You cannot use `memcmp()` to compare padded structures defined in this way.

Figure 6-1: Conventional structure example



sizeof(example) == 12 (3 words)

Packed structures

A **__packed struct** is one in which:

- the alignment of the packed structure, and of the fields within it, is always 1, independent of the alignment specified by `-zas`
- there is no padding between fields to ensure the natural alignment of each field
- there is no trailing padding to ensure the natural alignment of a following **struct** within an array.

Packed structures are defined with the **__packed** qualifier. Refer to *Type qualifiers* on page 4-35 for more information. Note the following important points:

- **float** and **double** types cannot be packed in a packed struct, or pointed to.
- Any variables declared using a packed tag automatically inherit the packed attribute, so **__packed** does not have to be specified.
- A **__packed struct** can never be assignment-compatible with an unpacked **struct**. It is impossible to assign a packed structure to a non-packed structure except by using field by field copy.
- The effect of casting away **__packed** is undefined.

The compiler generates code to perform unaligned access for pointers to packed types, when accessing packed elements, or when accessing a packed structure.

Note *On ARM processors, access to unaligned data can be expensive, taking up to seven instructions and two extra work registers. Data accesses through packed structures should be minimized to avoid performance loss. Generally, internal data structures should not be packed.*

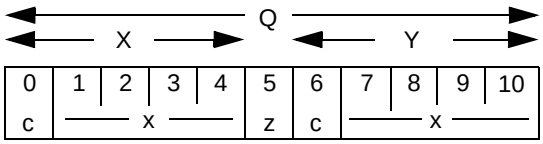
Sub-structs of packed structs

A **struct** sub-field of a packed **struct** or **union** must be declared to have **__packed struct** type. Similarly a **union** sub-field of a packed **struct** or **union** must be declared to have **__packed union** type. For example:

```
struct S {...};
__packed struct P {...};
struct T {
               struct S ss; // OK
               struct P pp; // OK
};
__packed struct Q {
               struct S ss; // faulted - sub-structs must be packed
               struct P pp; // OK
};
```

Figure 6-2: Sub-struct padding

```
__packed struct P {char c; int x;};
__packed struct {
    struct P X;
    char z;
    struct P Y;
} Q
```



Note: The structure contains no padding.

The sub-structs are abutted without any intermediate padding, and they contain no internal padding themselves because they must be packed.

Bitfields

ARM C++ compilers handle bitfields in non-packed structures in the following way.

Bitfields are allocated in *containers*. A container is a correctly aligned object of a declared type. Bitfields are allocated so that the first field specified occupies the lowest-addressed bits of the word, depending on configuration:

little-endian lowest addressed means least significant

big-endian lowest addressed means most significant.

A bitfield container may be any of the integral types, except that **long long** bitfields are not supported in big-endian mode. A plain bitfield, declared without either **signed** or **unsigned** qualifiers, is treated as **unsigned**, except in PCC mode where it is **signed**. For example, `int x:10` allocates an unsigned integer of 10 bits.

A bitfield is allocated to the first container of the correct type that has a sufficient number of unallocated bits. For example:

```
struct X {
               int x:10;
               int y:20;

};
```

The first declaration allocates an integer container in which 10 bits are allocated. At the second declaration, the compiler finds the existing integer container, checks that the number of unallocated bits are sufficient and allocates y in the same container as x.

A bitfield is wholly contained within its container. A bitfield that does not fit in a container is placed in the next container of the same type. For example, if an additional bitfield is declared for the structure above:

```
struct X {
               int x:10;
               int y:20;
               int z:5;
};
```

the declaration of z overflows the container. The compiler pads the remaining two bits for the first container and assigns a new integer container for z.

Bitfield containers may *overlap* each other. For example:

```
struct X{
               int x:10;
               char y:2;
};
```

The first declaration allocates an integer container in which 10 bits are allocated. These 10 bits occupy the first byte, and two bits of the second byte of the integer container. At the second declaration, the compiler checks for a container of type **char**. There is no suitable container, so the compiler allocates a new correctly aligned **char** container.

Because the natural alignment of **char** is 1, the compiler searches for the first byte that contains a sufficient number of unallocated bits to completely contain the bitfield. In the above example, the second byte of the **int** container has two bits allocated to `x`, and six bits unallocated. The compiler allocates a **char** container starting at the second byte of the previous **int** container, skips the first two bits that are allocated to `x`, and allocates two bits to `y`.

If `y` is declared `char y:8`:

```
struct X{
               int x:10;
               char y:8;
}
```

the compiler pads the second byte and allocates a new **char** container to the third byte, because the bitfield cannot overflow its container.

Note that the same basic rules apply to bitfield declarations with different container types. For example, adding an **int** bitfield to the example above:

```
struct X{
               int x:10;
               char y:8;
               int z:5;
}
```

The compiler allocates an **int** container starting at the same location as the `int x:10` container, and allocates a 5-bit bitfield. The structure as a whole looks like this:

`x`	a 10-bit bitfield
`padding`	of 6 bits
`y`	an 8-bit bitfield
`z`	a 5-bit bitfield
`unallocated`	3 unallocated bits.

Note that the above examples apply to bitfields in non-packed structures. The following applies to bitfields in packed structures.

Packed structures are structures without any byte padding, including any padding implied by the `-zas` compiler option. Bitfield containers in packed structures have an alignment of 1. This means that the maximum bit padding for a bitfield in a packed structure is 7 bits. For an unpacked structure, the maximum padding is `8*sizeofcontainer-1 bits`. Refer to *Packed structures* on page 6-11 for more information.

Structures can contain bitfields and non-bitfields in any order. The following rules apply:

- When a bitfield is followed by a non-bitfield in a structure, the bitfield is padded to the next available byte - not to the natural alignment of the container type.
- Bitfields can overlap non-bitfields. When a non-bitfield is followed by a bitfield in a structure, a new correctly aligned container is created that includes the first byte after the last non-bitfield, but is otherwise as empty as possible. For example:

  ```
  int x:10;
  char y;
  int z:5;
  ```

  results in exactly the same structure as:

  ```
  int x:10;
  char y:8;
  int z:5;
  ```

- Bitfields with a size equal to their container type behave as if they are not bitfields.

You can explicitly pad a bitfield container by declaring a bitfield of size zero. A bitfield of zero size fills the container up to the end if the container is non-empty. A subsequent bitfield declaration will start a new container.

6.5 Standard C++ Implementation Definition

This section gives details of those aspects of the ARM C++ implementation that the Draft Standard identifies as implementation defined.

In addition, Appendix A.6 of the ISO C standard collects together information about portability issues and section A.6.3 lists those points that must be defined by each implementation of ANSI C. When used in ANSI C mode, the ARM C++ compiler is identical to the ARM C compiler provided as part of the ARM Software Development Toolkit Version 2.11.

Refer to section 1.6 of the SDT 2.11 *Reference Guide* for detailed information on ARM C++ implementation of ANSI C.

6.5.1 Integral conversion (section 4.7 of the Draft Standard)

During integral conversion, if the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width). Otherwise, the value is truncated to fit the size of the destination type and a warning is given.

6.5.2 Standard C++ library implementation definition

For information on implementation-defined behavior that is defined in the Standard C++ library, refer to the Rogue Wave HTML documentation that is included with this release of ARM C++. By default, this is installed in the `/HTML` directory of your ARM 2.11 installation directory.

6.5.3 Language extensions

In ANSI C mode, the ARM C++ compiler implements the same extensions to ANSI C as the ARM C compiler. Refer to section 1.7 of the SDT 2.11 *Reference Guide* for information.

The ARM C++ compiler implements the following extensions to C++ as it is defined by the Draft Standard.

Long long

ARM C++ supports 64-bit integer types through the type specifier **long long**. **long long int** and **unsigned long long int** are integral types. They behave analogously to **long** and **unsigned long int** with respect to the usual arithmetic conversions.

Integer constants may have:

- an `LL` suffix to force the type of the constant to **long long**, if it will fit, or to **unsigned long long** if it will not
- an `LLU` (or `ULL`) suffix to force to the constant to **unsigned long long.**

Format specifiers for `printf()` and `scanf()` may include `LL` to specify that the following conversion applies to an (unsigned) **long long** argument, as in `%lld`.

In addition, a plain integer constant is of type (unsigned) **long long** if its value is large enough. This is a quiet change. For example in strict ANSI C, 2147483648 has type **unsigned long**. In ARM C++ it has the type **long long**. Thus the value of the expression `2147483648 > -1` is 1 in strict ANSI C and 0 with ARM C++.

The following restrictions apply to **long long**:

- **long long** bitfields are not supported in big-endian mode.
- **long long** enumerators are not available.
- The controlling expression of a switch statement may not have (unsigned) **long long** type. Consequently case labels must also have values that can be contained in a variable of type **unsigned long**.

Long Double

ARM C++ supports the **long double** type. However, it is implemented in the same manner as **double**.

6.6 Predefined Macros

Table 6-5 lists the macro names predefined by the ARM C++ compilers. Where the value field is empty, the symbol concerned is merely defined, as though by (for example) `-D__arm` on the command line.

Table 6-5: Predefined macros

Name	Value	Notes
`__STDC__`	1	defined in all compiler modes except PCC mode
`__cplusplus`	1	defined in C++ (default) compiler mode
`__CFRONT_LIKE`	1	defined in `-cfront` compiler mode
`__arm`		defined if using armcc, tcc, armcpp, or tcpp
`__thumb`		defined if using tcc or tcpp
`__SOFTFP`		defined if compiling to use the software floating-point library (`-apcs /softfp`)
`__APCS_NOSWT`		defined if `-apcs /noswst` in use
`__APCS_REENT`		defined if `-apcs /reent` in use
`__APCS_INTERWORK`		defined if `-apcs /interwork` in use
`__APCS_32`		defined unless `-apcs /26bit` is in use
`__APCS_NOFP`		defined if `-apcs /nofp` in use (no frame pointer)
`__PCS_FPREGARGS`		defined if `-apcs /fpregargs` is in use
`__BIG_ENDIAN`		defined if compiling for a big-endian target
`__TARGET_ARCH_xx`		*xx* represents the target architecture. The value of *xx* depends on the target architecture. For example, if the compiler option `-arch 4T` is specified then `__TARGET_ARCH_4T` is defined, and no other symbol starting with `_TARGET_ARCH_` is defined.
`__TARGET_CPU_xx`		*xx* represents the target cpu.The value of *xx* depends on the target cpu. For example, if the compiler option `-cpu ARM7TM` is specified then `_TARGET_CPU_ARM7TM` is defined and no other symbol starting with `_TARGET_CPU_` is defined. If the target architecture only is specified, without a target CPU then `_TARGET_CPU_generic` is defined.
`__TARGET_FEATURE_HALFWORD`		defined if the target architecture supports halfword and signed byte access instructions.
`__TARGET_FEATURE_MULTIPLY`		defined if the target architecture supports the long multiply instructions `MULL` and `MULAL`.
`__TARGET_FEATURE_THUMB`		defined if the target architecture is Thumb-aware.

Name	Value	Notes
`__ARMCC_VERSION`		Gives the version number of the compiler. The value is the same for armcc and tcc; it is a decimal number, whose value can be relied on to increase monotonically between releases.
`__CLK_TCK`	100	centisecond clock definition
`__CC_NORCROFT`	1	set by all Codemist compilers
`__sizeof_int`	4	`sizeof(int)`, but available in preprocessor expressions
`__sizeof_long`	4	`sizeof(long)`, but available in preprocessor expressions
`__sizeof_ptr`	4	`sizeof(void *)`, but available in preprocessor expressions
`__FILE__`		the presumed name of the current source file
`__LINE__`		the line number of the current source file
`__DATE__`		the date of translation of the source file
`__TIME__`		the time of translation of the source file

6.7 Implementation Limits

This section lists implementation limits for the ARM C++ compiler.

6.7.1 Draft Standard Limits

The Draft Standard standard sets out certain minimum limits that a conforming compiler must accept. You should be aware of these when porting applications between compilers. A summary is given in Table 6-6. The “mem” limit indicates that no limit is imposed by ARM C++, other than that imposed by the availability of memory.

Table 6-6: Implementation limits

Description	Required	ARM
Nesting levels of compound statements, iteration control structures, and selection control structures.	256	mem
Nesting levels of conditional inclusion.	256	mem
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration.	256	mem
Nesting levels of parenthesized expressions within a full expression.	256	mem
Number of initial characters in an internal identifier or macro name.	1024	mem
Number of initial characters in an external identifier.	1024	mem
External identifiers in one translation unit.	65536	mem
Identifiers with block scope declared in one block.	1024	mem
Macro identifiers simultaneously defined in one translation unit.	65536	mem
Parameters in one function declaration. Note that overload resolution is sensitive to the first 32 arguments only.	256	mem
Arguments in one function call. Note that overload resolution is sensitive to the first 32 arguments only.	256	mem
Parameters in one macro definition.	256	mem
Arguments in one macro invocation.	256	mem
Characters in one logical source line.	6536	mem
Characters in a characters string literal or wide string literal after concatenation.	65536	mem
Size of a C++ object.	262144	8388607
Nesting levels of #include file.	256	mem
Case labels for a switch statement, excluding those for any nested switch statements.	16384	mem
Data members ina single class, structure, or union.	16384	mem
Enumeration constants in a single enumeration.	4096	mem
Levels of nested class, structure, or union definitions in a single struct-declaration-list.	256	mem
Functions registered by `atexit()`.	32	33

Description	Required	ARM
Direct and indirect base classes	16384	mem
Direct base classes for a single class	1024	mem
Members declared in a single class	4096	mem
Final overriding virtual functions in a class, accessible or not	16384	mem
Direct and indirect virtual bases of a class	1024	mem
Static members of a class	1024	mem
Friend declarations in a class	4096	mem
Access control declarations in a class	4096	mem
Member initializers in a constructor definition	6144	mem
Scope qualifications of one identifier	256	mem
Nested external specifications	1024	mem
Template arguments in a template declaration	1024	mem
Recursively nested template instantiations	17	mem
Handlers per try block	256	mem
Throw specifications on a single function declaration	256	mem

6.7.2 Internal limits

In addition to the limits described in Table 6-6 on page 6-19, the compiler has the following internal limits:

Table 6-7: Internal limits

Description	ARM
Maximum number of relocatable references in a single translation unit.	65536
Maximum number of virtual registers.	65536
Maximum number of overload arguments.	32
Number of characters in a mangled name before it may be truncated.	128
Number of bits in the smallest object that is not a bit field (`CHAR_BIT`).	8
Maximum number of bytes in a multibyte character, for any supported locale (`MB_LEN_MAX`).	1

6.8 Limits for integral numbers

The following table gives the ranges for integral numbers as implemented in ARM C++. The third column of the table gives the numerical value of the range endpoint. The right hand column gives the bit pattern (in hexadecimal) that would be interpreted as this value in ARM C++.

When entering constants, you must be careful about the size and signed-ness of the quantity. Constants are interpreted differently in decimal and hexadecimal/octal. See the Draft Standard, or any of the recommended textbooks on the C++ programming language for more details.

Table 6-8: Integer ranges

Constant	Meaning	End-point	Hex Value
CHAR_MAX	Maximum value of **char**	255	0xff
CHAR_MIN	Minimum value of **char**	0	0x00
SCHAR_MAX	Maximum value of **signed char**	127	0x7f
SCHAR_MIN	Minimum value of **signed char**	-128	0x80
UCHAR_MAX	Maximum value of **unsigned char**	255	0xff
SHRT_MAX	Maximum value of **short**	32767	0x7fff
SHRT_MIN	Minimum value of **short**	-32768	0x8000
USHRT_MAX	Maximum value of **unsigned short**	65535	0xffff
INT_MAX	Maximum value of **int**	2147483647	0x7fffffff
INT_MIN	Minimum value of **int**	-214783648	0x80000000
LONG_MAX	Maximum value of **long**	2147483647	0x7fffffff
LONG_MIN	Minimum value of **long**	-214783648	0x80000000
ULONG_MAX	Maximum value of **unsigned long**	4294967295	0xffffffff

6.9 Limits for floating-point numbers

The following tables give the characteristics, ranges, and limits for floating-point numbers as implemented in ARM C++. Note also:

- when a floating-point number is converted to a shorter floating-point number, it is rounded to the nearest representable number
- the properties of floating-point arithmetic accord with IEEE 754.

Table 6-9: Floating-point limits

Constant	Meaning	Value
`FLT_MAX`	Maximum value of **float**.	3.40282347e+38F
`FLT_MIN`	Minimum value of **float**.	1.17549435e-38F
`DBL_MAX`	Maximum value of **double**.	1.79769313486231571e+308
`DBL_MIN`	Minimum value of **double**.	2.22507385850720138e-308
`LDBL_MAX`	Maximum value of **long double**.	1.79769313486231571e+308
`LDBL_MIN`	Minimum value of **long double**.	2.22507385850720138e-308
`FLT_MAX_EXP`	Maximum value of base 2 exponent for type **float**.	128
`FLT_MIN_EXP`	Minimum value of base 2 exponent for type **float**.	-125
`DBL_MAX_EXP`	Maximum value of base 2 exponent for type **double**.	1024
`DBL_MIN_EXP`	Minimum value of base 2 exponent for type **double**.	-1021
`LDBL_MAX_EXP`	Maximum value of base 2 exponent for type **long double**.	1024
`LDBL_MIN_EXP`	Minimum value of base 2 exponent for type **long double**.	-1021
`FLT_MAX_10_EXP`	Maximum value of base 10 exponent for type **float**.	38
`FLT_MIN_10_EXP`	Minimum value of base 10 exponent for type **float**.	-37
`DBL_MAX_10_EXP`	Maximum value of base 10 exponent for type **double**.	308
`DBL_MIN_10_EXP`	Minimum value of base 10 exponent for type **double**.	-307
`LDBL_MAX_10_EXP`	Maximum value of base 10 exponent for type **long double**.	308
`LDBL_MIN_10_EXP`	Minimum value of base 10 exponent for type **long double**.	-307

Table 6-10: Other floating-point characteristics

Constant	Meaning	Value
FLT_RADIX	Base (radix) of the ARM floating-point number representation.	2
FLT_ROUNDS	Rounding mode for floating-point numbers.	1 (nearest)
FLT_DIG	Decimal digits of precision for **float**.	6
DBL_DIG	Decimal digits of precision for **double**.	15
LDBL_DIG	Decimal digits of precision for **long double**.	15
FLT_MANT_DIG	Binary digits of precision for type **float**.	24
DBL_MANT_DIG	Binary digits of precision for type **double**.	53
LDBL_MANT_DIG	Binary digits of precision for type **long double**.	53
FLT_EPSILON	Smallest positive value of x such that 1.0 + x != 1.0 for type **float**.	1.19209290e-7F
DBL_EPSILON	Smallest positive value of x such that 1.0 + x != 1.0 for type **double**.	2.2204460492503131e-16
LDBL_EPSILON	Smallest positive value of x such that 1.0 + x != 1.0 for type **long double**.	2.2204460492503131e-16L

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

U

V

W

Z