ARM Software Development Toolkit

Version 2.50

Reference Guide

ARM

ARM DUI 0041C

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
Jan 1997	A	Created from ARM DUI 0020. Includes major updates for SDT 2.10
June 1997	B	Updated for SDT 2.11
Nov 1998	C	Updated for SDT 2.50

Proprietary Notice

ARM, Thumb, StrongARM, and the ARM Powered logo are registered trademarks of ARM Limited.

Angel, ARMulator, EmbeddedICE, Multi-ICE, ARM7TDMI, ARM9TDMI, and TDMI are trademarks of ARM Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Preface

This preface introduces the ARM Software Development Toolkit and its reference documentation. It contains the following sections:

- *About this book* on page iv
- *Further reading* on page vi
- *Typographical conventions* on page viii
- *Feedback* on page ix.

About this book

This book provides reference information for the ARM Software Development Toolkit. It describes the command-line options to the assembler, linker, compilers and other ARM tools, and gives reference material on software included with the Toolkit, such as the ARMulator.

Organization

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM Software Development Toolkit version 2.5, and details of the changes that have been made since version 2.11a

Chapter 2 *The ARM Compilers*

Read this chapter for an explanation of all command-line options accepted by the ARM C and C++ compilers.

Chapter 3 *ARM Compiler Reference*

Read this chapter for a description of the language features provided by the ARM C and C++ compilers, and for information on standards conformance and implementation details.

Chapter 4 *The C and C++ Libraries*

Read this chapter for a description of how to retarget and rebuild the ARM C and C++ libraries.

Chapter 5 *Assembler*

Read this chapter for an explanation of all command-line options accepted by the ARM assembler. In addition, this chapter documents features such as the directives and pseudo-instructions supported by the assembler.

Chapter 6 *Linker*

Read this chapter for an explanation of all command-line options accepted by the linker, and for reference information on linker features such as scatter loading.

Chapter 7 *ARM Symbolic Debugger*

Read this chapter for an explanation of all command-line options accepted by the ARM symbolic debugger.

Chapter 8 *Toolkit Utilities*

Read this chapter for a description of the utility programs provided with the ARM Software Development Toolkit, including fromELF, the ARM profiler, the ARM librarian, and the ARM object file decoders.

Chapter 9 *ARM Procedure Call Standard*

This chapter defines the ARM Procedure Call Standard (APCS).

Chapter 10 *Thumb Procedure Call Standard*

This chapter defines the Thumb Procedure Call Standard (TPCS).

Chapter 11 *Floating-point Support*

Read this chapter for reference information on floating-point support in the Software Development Toolkit.

Chapter 12 *ARMulator*

Read this chapter for reference material relating the the ARMulator.

Chapter 13 *ARM Image Format*

Read this chapter for a description of the AIF file format.

Chapter 14 *ARM Object Library Format*

Read this chapter for a description of the ALF file format.

Chapter 15 *ARM Object Format*

Read this chapter for a description of the AOF file format.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing for the ARM processor, and general information on related topics such as C and C++ development.

ARM publications

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

- *ARM Software Development Toolkit User Guide* (ARM DUI 0040)
- *ARM Architectural Reference Manual* (ARM DUI 0100)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- *ARM Target Development System User Guide* (ARM DUI 0061)
- the ARM datasheet for your hardware device.

Other publications

This book is not intended to be an introduction to the ARM assembly language, C, or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards. The following texts provide general information:

ARM architecture

- Furber, S., *ARM System Architecture* (1996). Addison Wesley Longman, Harlow, England. ISBN 0-201-40352-8.

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

The following books provide general C programming information:

- 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.

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.

<u>`type`</u>`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 where the argument is to be replaced by a specific value.

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, and for ARM processor signal names.

`typewriter bold`

Denotes language keywords when used outside example code.

Feedback

ARM Limited welcomes feedback on both the Software Development Toolkit, and the documentation.

Feedback on this book

If you have any comments on this book, please send email to errata@arm.com giving:

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

General suggestions for additions and improvements are also welcome.

Feedback on the ARM Software Development Toolkit

If you have any problems with the ARM Software Development Kit, 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
Reference Guide

Chapter 5 **Assembler**

Chapter 6 **Linker**

Chapter 7 **ARM Symbolic Debugger**

Chapter 8 **Toolkit Utilities**

Chapter 1
Introduction

This chapter introduces the ARM Software Development Toolkit version 2.50 and describes the changes that have been made since SDT version 2.11a. It contains the following sections:

- *About the ARM Software Development Toolkit* on page 1-2
- *Supported platforms* on page 1-5
- *What is new?* on page 1-6.

1.1 About the ARM Software Development Toolkit

The *ARM Software Development Toolkit* (SDT) consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

You can use the SDT to develop, build, and debug C, C++, or ARM assembly language programs.

1.1.1 Components of the SDT

The ARM Software Development Toolkit consists of the following major components:

- command-line development tools
- Windows development tools
- utilities
- supporting software.

These are described in more detail below.

Command-line development tools

The following command-line development tools are provided:

armcc The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI or PCC source into 32-bit ARM code.

tcc The Thumb C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI or PCC source into 16-bit Thumb code.

armasm The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source.

armlink The ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ELF executable images.

armsd The ARM and Thumb symbolic debugger. This enables source level debugging of programs. You can single step through C or assembly language source, set breakpoints and watchpoints, and examine program variables or memory.

Windows development tools

The following windows development tools are provided:

ADW The ARM Debugger for Windows. This provides a full Windows environment for debugging your C, C++, and assembly language source.

APM The ARM Project Manager. This is a graphical user interface tool that automates the routine operations of managing source files and building your software development projects. APM helps you to construct the environment, and specify the procedures needed to build your software.

Utilities

The following utility tools are provided to support the main development tools:

fromELF The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats, including AIF, plain binary, *Extended Intellec Hex* (IHF) format, Motorola 32-bit S record format, and Intel Hex 32 format.

armprof The ARM profiler displays an execution profile of a program from a profile data file generated by an ARM debugger.

armlib The ARM librarian enables sets of AOF files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several AOF files.

decaof The ARM Object Format decoder decodes AOF files such as those produced by armasm and armcc.

decaxf The ARM Executable Format decoder decodes executable files such as those produced by armlink.

topcc The ANSI to PCC C Translator helps to translate C programs and headers from ANSI C into PCC C, primarily by rewriting top-level function prototypes.

topcc is available for UNIX platforms only, not for Windows.

Supporting software

The following support software is provided to enable you to debug your programs, either under emulation, or on ARM-based hardware:

ARMulator The ARM core emulator. This provides instruction accurate emulation of ARM processors, and enables ARM and Thumb executable programs to be run on non-native hardware. The ARMulator is integrated with the ARM debuggers.

Angel The ARM debug monitor. Angel runs on target development hardware and enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.

1.1.2 Components of C++ version 1.10

ARM C++ is not part of the base Software Development Toolkit. It is available separately. Contact your distributor or ARM Limited if you want to purchase ARM C++.

ARM C++ version 1.10 consists of the following major components:

armcpp This is the ARM C++ compiler. It compiles draft-conforming C++ source into 32-bit ARM code.

tcpp This is the Thumb C++ compiler. It compiles draft-conforming C++ source into 16-bit Thumb code.

support software

The ARM C++ release provides a number of additional components to enable support for C++ in the ARM Debuggers, and the ARM Project Manager.

Note

The ARM C++ compilers, libraries, and enhancements to the ARM Project Manager and ARM Debuggers are described in the appropriate sections of the ARM Software Development Toolkit User Guide and Reference Guide.

1.2 Supported platforms

This release of the ARM Software Development Toolkit supports the following platforms:

- Sun workstations running Solaris 2.5 or 2.6
- Hewlett Packard workstations running HP-UX 10
- IBM compatible PCs running Windows 95, Windows 98, or Windows NT 4.

The Windows development tools (ADW and APM) are supported on IBM compatible PCs running Windows 95, Windows 98, and Windows NT 4.

The SDT is *no longer* supported on the following platforms:

- Windows NT 3.51
- SunOS 4.1.3
- HP-UX 9
- DEC Alpha NT.

1.3 What is new?

This section describes the major changes that have been made to the Software Development Toolkit since version 2.11a. The most important new features are:

- Improved support for debug of optimized code.
- Instruction scheduling compilers.
- Reduced debug data size.
- New supported processors. ARMulator now supports the latest ARM processors.
- ADW enhancements. SDT 2.50 provides a new ADW capable of remote debugging with Multi-ICE, and able to accept DWARF 1 and DWARF 2 debug images.

The preferred and default debug table format for the SDT is now DWARF 2. The ASD debug table format is supported for this release, but its use is deprecated and support for it will be withdrawn in future ARM software development tools.

The preferred and default executable image format is now ELF. Refer to the ELF description in `c:\ARM250\PDF\specs` for details of the ARM implementation of standard ELF format.

Demon-based C libraries are no longer included in the toolkit release, and RDP is no longer supported as a remote debug protocol.

The following sections describe the changes in more detail:

- *Functionality enhancements and new functionality* on page 1-7
- *Changes in default behavior* on page 1-12
- *Obsolete and deprecated features* on page 1-15.

1.3.1 Functionality enhancements and new functionality

This release of the ARM Software Development Toolkit introduces numerous enhancements and new features. The major changes are described in:

- *Improved support for debug of optimized code* on page 1-7
- *Instruction scheduling compilers* on page 1-8
- *Reduced debug data size* on page 1-8
- *New supported processors* on page 1-9
- *ADW enhancements* on page 1-9
- *Interleaved source and assembly language output* on page 1-10
- *New assembler directives and behavior* on page 1-10
- *Long long operations now compile inline* on page 1-11
- *Angel enhancements* on page 1-11
- *ARMulator enhancements* on page 1-11
- *New fromELF tool* on page 1-12
- *New APM configuration dialogs* on page 1-12.

Improved support for debug of optimized code

Compiling for debug (`-g`), and the optimization level (`-O`), have been made orthogonal in the compilers.

There are 3 levels of optimization:

`-O0` Turns off all optimization, except some simple source transformations.

`-O1` Turns off structure splitting, range splitting, cross-jumping, and conditional execution optimizations. Also, no debug data for inline functions is generated.

`-O2` Full optimization.

The `-O0` option gives the best debug view, but with the least optimized code.

The `-O1` option gives a satisfactory debug view, with good code density. By default no debug data is emitted for inline functions, so they cannot be debugged. With DWARF1 debug tables (`-dwarf1` command-line option), variables local to a function are not visible, and it is not possible to get a stack backtrace.

The `-O2` option emits fully optimized code that is still acceptable to the debugger. However, the correct values of variables are not always displayed, and the mapping of object code to source code is not always clear, because of code re-ordering.

A new pragma has been introduced to specify that debug data is to be emitted for inline functions. The pragma is `#pragma [no]debug_inlines`. You can use this pragma to bracket any number of inline functions. It can be used regardless of the level of optimization chosen.

Impact

Any existing makefiles or APM projects that use `–g+ –gxo` will now get the behavior defined by `–g+ -O1`. The SDT 2.11a option `-g+ -gxr` is still supported by SDT 2.50, and has the same functionality as in SDT 2.11a, but will not be supported by future releases.

Instruction scheduling compilers

The compilers have been enhanced to perform instruction scheduling. Instruction scheduling involves the re-ordering of machine instruction to suit the particular processor for which the code is intended. Instruction scheduling in this version of the C and C++ compilers is performed after the register allocation and code generation phases of the compiler.

Instruction scheduling is of benefit to code for the StrongARM1 and ARM9 processor families:

- if the `-processor` option specifies any processor other than the StrongARM1, instruction scheduling suitable for the ARM 9 is performed
- if `-processor StrongARM1` is specified, instruction scheduling for the StrongARM1 is performed.

By default, instruction scheduling is turned on. It can be turned off with the `-zpno_optimize_scheduling` command-line option.

Reduced debug data size

In SDT 2.50 and C++ 1.10, the compilers generate one set of debug areas for each input file, including header files. The linker is able to detect multiple copies of the set of debug areas corresponding to an input file that is included more than once, and emits only one such set of debug areas in the final image. This can result in a considerable reduction in image size. This improvement is not available when ASD debug data is generated.

In SDT 2.11a and C++ 1.01 images compiled and linked for debug could be considerably larger than expected, because debug data was generated separately for each compilation unit. The linker emitted all the debug areas, because it was unable to identify multiple copies of debug data belonging to header files that were included more than once.

Impact

Existing makefiles and APM projects generate smaller debug images, and therefore the images load more quickly into the debugger. This feature cannot be disabled.

New supported processors

ARMulator models for the ARM9TDMI, ARM940T, ARM920T, ARM710T, ARM740T, ARM7TDMI-S, ARM7TDI-S, and ARM7T-S processors have been added to SDT 2.50. These are compatible with the memory model interfaces from the SDT 2.11a ARMulator.

These processor names (and those of all other released ARM processors) are now permitted as arguments to the `–processor` command-line option of the compilers and assembler.

ADW enhancements

ADW has been enhanced to provide the following additional features:

- Support for remote debug using Multi-ICE.
- Support for reading DWARF 2 debug tables.
- The command-line options supported by armsd that are suitable for a GUI debugger are now understood on the ADW command line. This enables you, for example, always to start ADW in remote debug mode. The available command-line options are:
  - `-symbols`
  - `-li, -bi`
  - `-armul`
  - `-adp –linespeed` *`baudrate`* `-port` `[s=`*`serial_port`*`[,p=`*`parallel_port`*`]] |` `[e=`*`ethernet_address`*`]`
- A *delete all breakpoints* facility.
- Save and restore all window formats. Windows retain the format they were given.
- Breakpoints can be set as 16-bit or 32-bit. The dialog box for setting a breakpoint has been modified to enable breakpoints to be set either as ARM or Thumb breakpoints, or for the choice to be left to the debugger.
- The display of low-level symbols can be sorted either alphabetically or by address order (sorting was by address order only in SDT 2.11a). You can choose the order that is used.

- Locals, Globals, and Debugger Internals windows format is now controlled by `$int_format`, `$uint_format`, `$float_format`, `$sbyte_format`, `$ubyte_format`, `$string_format`, `$complex_format`. These formats are available by selecting **Change Default Display Formats** from the **Options** menu.
- The Memory window now has *halfword* and *byte* added to its display formats.
- Value fields in editable windows (for example, Variable windows and Memory windows) are now *edit in place*, rather than using a separate dialog box for entering new values.

A copy of ADW is also supplied in a file named `MDW.exe` to maintain backwards compatibility with the Multi-ICE release.

Interleaved source and assembly language output

The compilers in SDT 2.50 and C++ 1.10 have been enhanced to provide an assembly language listing, annotated with the original C or C++ source that produced the assembly language. Use the command-line options `-S –fs` to get interleaved source and assembly language.

This facility is not available if ASD debug tables are requested (`-g+ -asd`).

This facility is only easily accessible from the command line, and is not integrated with APM.

New assembler directives and behavior

The SDT 2.11a assemblers (armasm and tasm) have been merged into a single assembler, called armasm, that supports both ARM code and Thumb code. In addition, it provides functionality previously supported only by tasm, such as the `CODE16` and `CODE32` directives, and the `-16` and `-32` command-line options. The assembler starts in ARM state by default. A tasm binary is shipped with SDT 2.50 for compatibility reasons, however this binary only invokes `armasm –16`.

The assembler now supports the following FPA pseudo-instructions:

- `LDFS fp-register, =fp-constant`
- `LDFD fp-register, =fp-constant`
- `LDFE fp-register, =fp-constant`

and the new directives `DCWU` and `DCDU`.

Long long operations now compile inline

In the C and C++ compilers, the implementation of the **`long long`** data type has been optimized to inline most operators. This results in smaller and faster code. In particular:

```
long long res = (long long) x * (long long) y;
```

translates to a single `SMULL` instruction, instead of a call to a **`long long`** multiply function, if x and y are of type **`int`**.

Angel enhancements

Angel has been enhanced to enable full debug of interrupt-driven applications.

ARMulator enhancements

The following enhancements have been made to the ARMulator:

- Total cycle counts are always displayed.
- Wait states and true idle cycles are counted separately if a map file is used.
- F bus cycle counts are displayed if appropriate.
- Verbose statistics are enabled by the line `Counters=True` in the `armul.cnf` file. For cached cores, this adds counters for TLB misses, write buffer stalls, and cache misses.
- The instruction tracer now supports both Thumb and ARM instructions.
- A new *fast* memory model is supplied, that enables fast emulation without cycle counting. This is enabled using `Default=Fast` in the `armul.cnf` file.
- Trace output can be sent to a file or appended to the RDI log window.

New fromELF tool

The fromELF translation utility is a new tool in SDT 2.50. It can translate an ELF executable file into the following formats:

- AIF family
- Plain binary
- Extended Intellec Hex (IHF) format
- Motorola 32 bit S record format
- Intel Hex 32 format
- Textual Information.

This tool does not have a GUI integrated with APM. It can be called directly from the command line, or by editing your APM project to call fromELF after it calls the linker.

New APM configuration dialogs

The Tool Configurer dialog boxes have been modified to reflect:

- the new features available in the compilers, assembler, and the linker
- the new default behavior of these tools.

Each selectable option on the dialog boxes now has a tool tip that displays the command-line equivalent for the option.

1.3.2 Changes in default behavior

The changes that have been made to the default behavior of the SDT are described in:

- *Stack disciplines* on page 1-12
- *Default Procedure Call Standard (APCS and TPCS)* on page 1-13
- *Default debug table format* on page 1-13
- *Default image file format* on page 1-14
- *Default processor in the compilers and assembler* on page 1-14
- *RDI 1.0 and RDI 1.5 support* on page 1-14
- *Register names permitted by the assembler* on page 1-15.

Stack disciplines

The ARM and Thumb compilers now adjust the stack pointer only on function entry and exit. In previous toolkits they adjusted the stack pointer on block entry and exit. The new scheme gives improved code size.

Default Procedure Call Standard (APCS and TPCS)

The default *Procedure Call Standard* (PCS) for the ARM and Thumb compilers, and the assembler in SDT 2.50 and C++ 1.10 is now:

```
-apcs 3/32/nofp/noswst/narrow/softfp
```

Note

The new default PCS will not perform software stack checking and does not use a frame pointer register. This generates more efficient and smaller code for use in embedded systems.

The default procedure call standard for the ARM (not Thumb) compiler in SDT 2.11a was `-apcs 3/32/fp/swst/wide/softfp`.

The default procedure call standard for the ARM (not Thumb) assembler in SDT 2.11a was `-apcs 3/32/fp/swst`.

Impact

Existing makefiles and APM project files where the PCS was not specified will generate code that does not perform software stack checking and does not use a frame pointer register. This will result in smaller and faster code, because the default for previous compilers was to emit function entry code that checked for stack overflow and set up a frame pointer register.

Default debug table format

In SDT 2.50 and C++ 1.10 the default debug table format is DWARF 2. DWARF 2 is required to support debugging C++, and to support the improvements in debugging optimized code.

The default debug table format emitted by the SDT 2.11a compilers and assemblers was ASD.

If DWARF debug table format was chosen, the SDT 2.11a compilers and assemblers emitted DWARF 1.0.3.

Impact

Existing makefiles and APM project files where debugging information was requested will now result in DWARF 2 debug data being included in the executable image file. Previous behavior can be obtained from the command line by specifying `-g+ -asd` or `-g+ -dwarf1`, or by choosing these from the appropriate Tool Configuration dialog boxes in APM.

Default image file format

The default image file format emitted by the linker has changed from AIF to ELF.

Impact

Existing makefiles in which no linker output format was chosen, and existing APM project files in which the **Absolute AIF** format was chosen, will now generate an ELF image. If you require an AIF format image, use `-aif` on your armlink command line, or choose **Absolute AIF** on the **Output** tab of the APM Linker options dialog box. This will then generate a warning from the linker. AIF images can also be created using the new fromELF tool.

Note

When the ARM debuggers load an executable AIF image they switch the processor mode to User32. For ELF, and any format other than executable AIF, the debuggers switch the processor mode to SVC32. This means that, by default, images now start running in SVC32 mode rather than User32 mode. This better reflects how the ARM core behaves at reset.

C code that performs inline SWIs must be compiled with the `-fz` option to ensure that the SVC mode link register is preserved when the SWI is handled.

Default processor in the compilers and assembler

The default processor for the SDT 2.11a ARM (not Thumb) compilers was ARM6. In SDT 2.50 and C++ 1.10 this has been changed to ARM7TDMI. The default processor for the assembler has changed from `-cpu generic –arch 3` to `-cpu ARM7TDMI`.

Impact

Existing makefiles and APM project files where the processor was not specified (with the `-processor` option) will generate code that uses halfword loads and stores (`LDRH`/`STRH`) where appropriate, whereas such instructions would not previously have been generated. Specifying `-arch 3` on the command line prevents the compilers from generating halfword loads and stores.

RDI 1.0 and RDI 1.5 support

A new variant of the Remote Debug Interface (RDI 1.5) is introduced in SDT 2.50. The version used in SDT 2.11a was 1.0.

The debugger has been modified so that it will function with either RDI 1.0 or RDI 1.5 client DLLs.

Impact

Third party DLLs written using RDI 1.0 will continue to work with the versions of ADW and armsd shipped with SDT 2.50.

Register names permitted by the assembler

In SDT 2.50, the assembler pre-declares all PCS register names, but also allows them to be declared explicitly through an `RN` directive.

In SDT 2.11a the procedure call standard (PCS) register names that the assembler would pre-declare were restricted by the variant of the PCS chosen by the `-apcs` option. For example, `-apcs /noswst` would disallow use of sl as a register name.

Impact

Any source files that declared PCS register names explicitly will continue to assemble without fault, despite the change to the default PCS.

1.3.3 Obsolete and deprecated features

The features listed below are either obsolete or deprecated. Obsolete features are identified explicitly. Their use is faulted in SDT 2.50. Deprecated features will be made obsolete in future ARM toolkit releases. Their use is warned about in SDT 2.50. These features are described in:

- *AIF, Binary AIF, IHF and Plain Binary Image formats* on page 1-16
- *Shared library support* on page 1-16
- *Overlay support* on page 1-16
- *Frame pointer calling standard* on page 1-17
- *Reentrant code* on page 1-17
- *ARM Symbolic Debug Table format (ASD)* on page 1-17
- *Demon debug monitor and libraries* on page 1-18
- *Angel as a linkable library, and ADP over JTAG* on page 1-18
- *ROOT, ROOT-DATA and OVERLAY keywords in scatter load description* on page 1-18
- *Automatically inserted ARM/Thumb interworking veneers* on page 1-18
- *Deprecated PSR field specifications* on page 1-19
- *ORG no longer supported in the assembler* on page 1-19.

AIF, Binary AIF, IHF and Plain Binary Image formats

Because the preferred (and default) image format for the SDT is now ELF, the linker emits a warning when instructed to generate an AIF image, a binary AIF image, an IHF image or a plain binary image.

Impact

Any makefiles with a link step of `-aif`, `-aif -bin`, `-ihf`, or `-bin` now produce a warning from the linker. For existing APM projects where an Absolute AIF image has been requested on the Linker configuration **Output** tab, there will be no warning. However, an ELF image is created instead, because this is the new default for the linker.

The preferred way to generate an image in an deprecated format is to create an ELF format image from the linker, and then to use the new fromELF tool to translate the ELF image into the desired format.

Future release

In a future release of the linker, these formats will be obsolete, and their use will be faulted.

Shared library support

This feature is obsolete. The Shared Library support provided by previous versions of the SDT has been removed for SDT 2.50. The linker faults the use of the `-shl` command-line option.

Impact

Any makefile or APM project file that uses the Shared Library mechanism will now generate an error from the linker. The SDT 2.11a linker can be used if this facility is required.

Future release

A new Shared Library mechanism will be introduced in a future release of the linker.

Overlay support

Use of the `-overlay` option to the linker and use of the `OVERLAY` keyword in a scatter load description file are now warned against by the linker.

Impact

Any makefile, APM project file, or scatter load description file that uses the overlay mechanism will now generate a warning from the linker.

Future release

A future release of the linker will subsume the overlay functionality into the scatter loading mechanism.

Frame pointer calling standard

Use of a frame pointer call standard when compiling C or C++ code is warned against in the SDT 2.50 and ARM C++ 1.10 versions of the compilers.

Impact

Any makefile or APM project file that uses a frame pointer call standard (`-apcs /fp`) will now generate a warning from the compilers.

Future release

A new procedure call standard will be introduced with a future release of the compilers.

Reentrant code

Use of the reentrant procedure call standard when compiling C or C++ code is warned against in the SDT 2.50 and ARM C++ 1.10 versions of the compilers.

Impact

Any makefile or APM project file that uses the reentrant procedure call standard (`-apcs /reent`) will now generate a warning from the compilers.

Future release

A new procedure call standard will be introduced with a future release of the compilers.

ARM Symbolic Debug Table format (ASD)

Because the preferred (and default) debug table format is now DWARF 2, the compilers and assembler will warn when asked to generate ASD debug tables.

Impact

Any makefiles with a compiler or assembler command-line option of `-g+ –asd` will now produce a warning. For existing APM projects in which debugging information has been requested, there will be no warning and DWARF 2 debug tables will be emitted instead, because this is the new default for the compilers and assembler.

Future release

In a future release of the compilers and assembler, ASD will be made obsolete, and its use will be faulted.

Demon debug monitor and libraries

This feature is obsolete. The Demon Debug monitor is now obsolete and support for it has been removed from the SDT. There is no longer a `remote_d.dll` selectable as a remote debug connection in ADW, and Demon C libraries are not supplied with SDT 2.50.

Angel as a linkable library, and ADP over JTAG

This feature is obsolete. Full Angel is no longer available as a library to be linked with a client application. The version of Angel that runs on an EmbeddedICE and acts as an ADP debug monitor (`adpjtag.rom`) is also no longer available.

ROOT, ROOT-DATA and OVERLAY keywords in scatter load description

In the SDT 2.11 manuals, use of the `ROOT`, `ROOT-DATA` and `OVERLAY` keywords in a scatter load description file was documented, and a later Application Note warned against its use. The linker now warns against use of these keywords.

Impact

Any existing scatter load descriptions that use `ROOT`, `ROOT-DATA` or `OVERLAY` keywords will now generate a warning, but the behavior will be as expected.

Future release

In a future release of the linker, use of `ROOT`, `ROOT-DATA` and `OVERLAY` will be faulted.

Automatically inserted ARM/Thumb interworking veneers

In SDT 2.11a, the linker warned of calls made from ARM code to Thumb code or from Thumb code to ARM code (interworking calls) when the destination of the call was not compiled for interworking with the `-apcs /interwork` option. In spite of the warning, an interworking return veneer was inserted.

In SDT 2.50, the linker faults inter-working calls to code that cannot return directly to the instruction set state of the caller, and creates no executable image.

Impact

Existing code that caused the interworking warning in SDT 2.11a is now faulted because the return veneers inserted by the SDT 2.11a linker can cause incorrect program behavior in obscure circumstances.

Deprecated PSR field specifications

The assembler now warns about the use of the deprecated field specifiers CPSR, CPSR_flg, CPSR_ctl, CPSR_all, SPSR, SPSR_flg, SPSR_ctl, and SPSR_all.

ORG no longer supported in the assembler

The `ORG` directive is no longer supported in the assembler. Its use conflicts with the scatter loading mechanism supported by the linker.

Impact

Existing assembly language sources that use the `ORG` directive will no longer assemble. The effect of the `ORG` directive can be obtained by using the scatter loading facility of the linker.

Chapter 2
The ARM Compilers

This chapter provides details of the command-line options to the ARM and Thumb, C and C++ compilers. It assumes that you are familiar with the basic concepts of using command-line software 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 User Guide*.

This chapter contains the following sections:

2.1 About the ARM compilers

The ARM C and 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 Windows/MS-DOS.

In addition, the ARM C++ compilers compile C++ that conforms to the ISO/IEC Draft Standard for C++ (December 1996). See *C++ language feature support* on page 3-49 for a detailed description of ARM C++ language support.

Note

The ARM C compilers are provided as standard with the Software Development Toolkit. The ARM C++ compilers are available separately. Contact your distributor or ARM Limited if you want to purchase the ARM C++ compilers.

2.1.1 Compiler variants

All ARM C and C++ compilers accept the same basic command-line options. The descriptions in this chapter apply to all compilers. Where specific compilers have added features or restrictions they are noted in the text. Where an option applies only to C++ this is also noted in the text.

There are two variants of the C compiler:

armcc compiles C source into 32-bit ARM code

tcc compiles C source into 16-bit Thumb code.

Throughout this chapter, armcc and tcc are referred to together as the ARM C compilers.

There are two variants of the C++ compiler:

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

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

Throughout this chapter, armcpp and tcpp are referred to together as the ARM C++ compilers.

2.1.2 Source language modes

The ARM 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:

ANSI C mode

In ANSI C mode, the ARM compilers are 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 generates 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.

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

Cfront mode

This mode applies only to the ARM C++ compilers. In Cfront mode, the ARM C++ compilers are more likely to accept 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 C++ programs that conform to the C++ 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 normally 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 class names 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`

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 2-16.

2.1.3 Compatibility between compilers

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

Note

Refer to the *ARM Software Development Toolkit User Guide* for information on how to link compiled ARM and Thumb code together.

2.1.4 Library support

The SDT provides ANSI C libraries in both source and binary form, and Rogue Wave C++ libraries in prebuilt binary form. Refer to Chapter 4 *The C and C++ Libraries* for detailed information about the libraries.

2.2 File usage

This section describes:

- the file naming conventions used by the ARM compilers
- how the ARM compilers search for `#include` header files and source files.

2.2.1 Naming conventions

The ARM compilers use suffix naming conventions to identify the classes of file involved in the compilation and linking processes. The names used on the command line, and as arguments to preprocessor `#include` directives, map directly to host file names under UNIX and Windows/MS-DOS.

The ARM compilers recognize the following file suffixes:

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

In addition, the ARM C++ compilers recognize 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

The ARM compilers support multiple file naming conventions on all supported hosts. Follow these guidelines:

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

In each host environment the compilers support:

- 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 that does not start with `host-volume-name:/` and contains `/` is a UNIX filename
- otherwise, the name is a host name.

Filename validity

The compilers do not check that filenames are acceptable to the host file 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 a compiler are stored in the current directory. Object files are written in ARM Object Format (AOF). AOF is described in Chapter 15 *ARM Object Format*.

2.2.2 Included files

There are a number of factors that affect how the ARM compilers search for `#include` header files and source files. These include:

- 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 file system.

The in-memory file system

The ARM compilers have the ANSI C library headers built into a special, textually compressed, in-memory file system. By default, the C header files are used from this file system. The in-memory file system can be specified explicitly on the command line as `-j-` and `-I-`.

In addition, the C++ header files that are equivalent to the C library header files are also stored in the in-memory file system. The Rogue Wave library header files, and the ARM-supplied headers such as `iostream.h` are not stored in the in-memory file system.

Enclosing a `#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 file system. For example:

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

Enclosing a `#include` filename in double quotes 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 2-17 for detailed information on how the ARM compilers search for include files.

The current place

By default, the ARM compilers adopt 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 `~/arm250/include` and the compiler is seeking the include file `sys/defs.h,` it will locate `~/arm250/include/sys/defs.h` if it exists.

When the compiler begins to process `defs.h`, the current place becomes `~/arm250/include/sys`. Any file included by `defs.h` that is not specified with an absolute pathname is sought relative to `~/arm250/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 2-1 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 the ARM compilers store ANSI C and some C++ header files. See *The in-memory file system* on page 2-7 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 2-8 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 before directories specified by `-j`, irrespective of their relative order on the command line. To specify the in-memory file system use `-I-`, or `-j-`.

Table 2-1 Include file search paths

Compiler Option	<include>	"include.h"
`not -I or -j`	`:mem, ARMINC`	`CP, ARMINC,:mem`
`-j`	`jdirs`	`CP, jdirs`
`-I`	`:mem, Idirs, ARMINC`	`CP, Idirs, ARMINC, :mem`

Table 2-1 Include file search paths (Continued)

Compiler Option	<include>	"include.h"
`both -I and -j`	`Idirs, jdirs`	`CP, Idirs, jdirs`
`-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.

2.3 Command syntax

This section describes the command syntax for the ARM C and 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. In most cases the ARM C and C++ compilers allow space between the option letter and the argument.

2.3.1 Invoking the compiler

The command for invoking the ARM compilers is:

```
compiler_name [PCS-options] [source-language] [search-paths]
[preprocessor-options] [output-format] [target-options]
[debug-options] [code-generation-options] [warning-options]
[additional-checks] [error-options] [miscellaneous-options]
source…
```

where:

compiler_name is one of armcc, tcc, armcpp, tcpp.

PCS-options specify the procedure call standard to use. See *Procedure Call Standard options* on page 2-14 for details.

source-language specifies the source language that is accepted by the compiler. The default is ANSI C for the C compilers and Draft Standard C++ for the C++ compilers. See *Setting the source language* on page 2-16 for details.

search-paths specify the directories that are searched for included files. See *Specifying search paths* on page 2-17 for details.

preprocessor-options

control aspects of the preprocessor, including preprocessor output and macro definitions. See *Setting preprocessor options* on page 2-18 for details.

output-format specifies the format for the compiler output. You can use these options to generate assembly language output (including interleaved source and assembly language), listing files and unlinked object files. See *Specifying output format* on page 2-19 for details.

target-options specify the target processor and architecture. See *Specifying the target processor and architecture* on page 2-20 for details.

`debug-options` specify whether or not debug tables are generated, and their format. See *Generating debug information* on page 2-23 for details.

`code-generation-options`

specify options such as optimization, endianness, and alignment of data produced by the compiler. See *Controlling code generation* on page 2-24 for details.

`warning-options` control whether specific warning messages are generated. See *Controlling warning messages* on page 2-29 for details.

`additional-checks`

specify a number of additional checks that can be applied to your code, such as checks for data flow anomalies and unused declarations. See *Specifying additional checks* on page 2-32 for details.

`error-options` allow you to:

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

See *Controlling error messages* on page 2-33 for details.

`pragma-options` allows you to emulate `#pragma` directives. See *Pragma emulation* on page 2-35 for details.

`source` are the filenames of one or more text files containing C or C++ source code, or - to specify keyboard input. 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 `-fp` can be combined as `-fph.` 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

The following option enables you to read additional command-line options from a file:

`-via` *`filename`*

Opens a file and reads additional command-line options from it. For example:

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

The options specified in *`filename`* are read at the same time as any other command-line options are parsed. If `-via` is specified in the Extra command line arguments text box of the APM Compiler Configuration dialog, the command-line options are immediately read into the tool configuration settings.

You can nest `-via` calls within `-via` files.

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 a MS Windows DOS environment.

An assembly listing for the function is sent to the output stream at the end of each C or C++ function if either:

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

If an output file is specified with the `-o` option, an object file is written. If the `-E` option is specified, the preprocessor output is sent to the output stream.

Getting help and version information

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

Use the `-vsn` compiler option to display the version string for the compiler.

Redirecting standard errors

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

2.3.2 Procedure Call Standard options

This section applies to both:

- the ARM Procedure Call Standard (APCS), used by armcc and armcpp
- the Thumb Procedure Call Standard (TPCS), used by tcc and tcpp.

Refer to Chapter 9 *ARM Procedure Call Standard* and Chapter 10 *Thumb Procedure Call Standard* for more information on the ARM and Thumb procedure call standards.

Use the following command-line options to specify the variant of the procedure call standard (PCS) that is to be used by the compiler:

`-apcs` *`qualifiers`*

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.

If no `-apcs` options are specified, the default for all compilers is:

```
-apcs /nofp/noswst/narrow/softfp/nonreentrant
```

The qualifiers are listed below.

Stack checking

`/swstackcheck`

Software stack-checking PCS variant.

`/noswstackcheck`

No software stack-checking PCS variant. This is the default.

Frame pointers

`/fp` Use a dedicated frame-pointer register. This option is obsolete and is provided for backwards compatibility only.

`/nofp` Do not use a frame-pointer register. This is the default.

Floating-point compatibility

`/softfp` Use software floating-point library functions for floating-point code. This is the default for ARM processors without an FPU, and the only floating-point option available in tcc or tcpp.

This option implies `-fpu none`. Use the -fpu option in preference to `/softfp` and `/hardfp`. See *Specifying the target processor and architecture* on page 2-20 for more information.

`/hardfp` Generate ARM coprocessor instructions for the FPA floating-point unit. You may also specify `/fpregargs` or `/nofpregargs`. The `/hardfp` and `/softfp` options are mutually exclusive. This option is not available in tcc or tcpp.

This option implies `-fpu fpa`. Use the `-fpu` option in preference to `/softfp` and `/hardfp`. See *Specifying the target processor and architecture* on page 2-20 for more information.

`/fpregargs` Floating-point arguments are passed in floating-point registers. This option is not available for in tcc or tcpp, and is the default for ARM if `/hardfp` is selected.

`/nofpregargs`

Floating-point arguments are not passed in floating-point registers. This option is not available in tcc or tcpp. This option is obsolete and is available for backwards compatibility only.

ARM/Thumb interworking

`/nointerwork`

Compile code for no ARM/Thumb interworking. This is the default.

`/interwork` Compile code for ARM/Thumb interworking. Refer to the *ARM Software Development Toolkit User Guide* for more information on ARM/Thumb interworking.

Narrow parameters

`/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.

This is the default. In PCC mode, `/narrow` only affects ANSI-style function prototypes. PCC-style functions always use callee-narrowing (see the `/wide` PCS option).

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

Reentrancy

`/reentrant` Reentrant PCS variant. This option is obsolete and is provided for backwards compatibility only. The SDT 2.5 linker will not link objects compiled with `-apcs /reentrant`.

`/nonreentrant`

Non-reentrant PCS variant. This is the default.

2.3.3 Setting the source language

This section describes options that have a major effect on the source language accepted by the compiler. The compiler modes are described in detail in *Source language modes* on page 2-3. See also, *Controlling code generation* on page 2-24.

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 C compilers attempt to compile ANSI C, and the C++ compilers attempt to compile C++ that conforms to the Draft Standard.

`-ansi` Compiles ANSI standard C. This is the default for armcc and tcc.

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

`-cfront` This option applies to the C++ compilers only.

The compiler alters its behavior so that it is more likely to accept C++ programs that Cfront accepts.

`-fussy` Is extra strict about enforcing conformance to the ANSI C standard, Draft 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 C++ 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, however the built-in ANSI C headers are still used. See also, the `-zc` option in *Controlling code generation* on page 2-24.

The `-pcc` option alters the language accepted by the compilers in the following ways:

- **`char`** is signed
- **`sizeof`** is signed
- an approximation of early UNIX-style C preprocessing is used.

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

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

2.3.4 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 *Included files* on page 2-7 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 file system is specified by `-I-`.

`-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 2-8 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. Use `-j-` to search the in-memory file system.

2.3.5 Setting preprocessor options

The following command-line options control aspects of the preprocessor. Refer to *Pragmas controlling the preprocessor* on page 3-3 for descriptions of other preprocessor options that can be set by pragmas.

`-E`
: Executes only the preprocessor phase of the compiler. By default, 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:

```
compiler-name -E source.c > rawc
```

You can also use the `-o` option to specify an output file.

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 2-19 for a description of the `-c` option.

`-M`
: Executes only the preprocessor phase of the compiler, as with `-E`. This option produces a list of makefile dependency lines suitable for use by a make utility.

By default, output is on the standard output stream. You can redirect output to a file by using standard UNIX/MS-DOS notation. For example:

```
compiler-name -M source.c >> Makefile
```

You can also specify an output file with the `-o` option.

`-MD`
: As with `-M`. This option writes a list of makefile dependency lines suitable for use by a make utility to *`inputfilename`*`.d`

`-MD-`
: This option applies to the ARM Project Manager. When specified in the APM build step command line it writes makefile dependencies to the invoking Project Manager.

`-D`*`symbol=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.

`-D`*`symbol`*
: 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.

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

```
#undef symbol
```

This option can be repeated.

2.3.6 Specifying output format

By default, source files are compiled and linked into an executable image. You can use the following options 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 2-18 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 2-18.

`-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`.

`-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 */
```

If `-fu` is not specified, this is listed as:

```
p = 0;
```

If `-fu` is specified it is listed as:

```
p = NULL;
```

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

- in conjunction with `-c`, it names the object file
- in conjunction with `-S`, it names the assembly language file
- in conjunction with `-E`, it specifies the output file for preprocessed source.
- if none of `-c` , `-S`, or `-E` 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 `-E` is specified output is sent to the standard output stream.

If none of `-c`, `-S`, or `-E` is specified, an executable image called `file2` is created.

If *`file`* is specified as `-`, output is sent to the standard output stream.

`-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 with the `-o` option.

`-fs` When used with -S, writes a file containing C or C++ source code interleaved line by line with the assembly language generated by the compiler from that source code.

2.3.7 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, such as support for halfword instructions and instruction scheduling. This may make the code incompatible with other ARM processors.

If neither a `-processor` or `-architecture` option is specified, the default is `-processor ARM7TDMI`.

The following general points apply to processor and architecture options:

- If you specify a `-architecture` option, the compiler compiles code that runs on any processor that supports that architecture.
  - To enable halfword load and store instructions, specify one of the architecture 4 options, or specify an appropriate processor using a `-processor` option such as ARM7TDMI, ARM9TDMI, or StrongARM1.

    - To enable long multiply, specify `-architecture 3M`, `-architecture 4`, `-architecture 4M`, or `-architecture 4T`, or specify an appropriate `-processor` option such as ARM7DMI.

- If you specify a `-processor` option, the compiler compiles code and optimizes specifically for that processor. The code may be incompatible with other ARM processors that support the same architecture.

  For example, a `-processor` option may imply the presence of specific coprocessors, or allow instruction scheduling for optimum performance on the specified processor.

- Specifying a Thumb-aware processor, such as `-processor ARM7TDMI,` to armcc or armcpp does not make these compilers generate Thumb code. It only allows features of the processor to be used, such as support for halfword instructions. You must use tcc or tcpp to generate Thumb code.

The following options are available:

`-architecture` *`n`*

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

- 3
- 3M
- 4
- 4M
- 4xM. This is architecture 4M without long multiply instructions.
- 4T
- 4TxM. This is architecture 4T without long multiply instructions.

It is not necessary to specify both `-architecture` and `-processor`. If a `-architecture` option is specified and a `-processor` option is not, the default processor is generic.

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

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

`none` No FPU. Use software floating point library. This option implies `/softfp`.

`fpa` Floating Point Accelerator. This option implies `/hardfp`.

`-processor` *`name`*

Compiles code for the specified ARM processor where *`name`* is the name of the ARM processor. It is not necessary to specify both `-architecture` and `-processor`. The `-processor` option is specific to an individual processor. The `-architecture` option compiles code for any processor that conforms to the specified architecture. Valid values are:

`ARM6`	Implements architecture 3.
`ARM7`	Implements architecture 3.
`ARM7M`	Implements architecture 3M.
`ARM7TM`	Implements architecture 4T.
`ARM7T`	Implements architecture 4TxM.
`ARM7TDI`	Implements architecture 4TxM.
`ARM7TDMI`	Implements architecture 4T. 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.
`ARM8`	Implements architecture 4.
`ARM9`	Implements architecture 4.
`ARM9TDMI`	Implements architecture 4T.
`SA-110`	Implements architecture 4.
`StrongARM1`	Implements architecture 4.
`generic`	This processor name is used when compiling for a specified architecture and no `-processor` option is specified.

2.3.8 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 *Pragmas controlling debugging* on page 3-4 for more information on controlling debug information.

Note

Optimization criteria may impose limitations on the debug information that can be generated by the compiler. Refer to *Defining optimization criteria* on page 2-25 for more information.

Debug table generation options

The following options specify how debug tables are generated:

`-g+` Switches the generation of debug tables on for the current compilation. Debug table options are as specified by `-gt`. Optimization options for debug code are as specified by `-O`. `-g` is a synonym for `-g+`, but its use is deprecated. By default, the `-g+` option is equivalent to:

```
-g+ -dwarf2 -O0 -gt
```

By default, debug data is not generated for **`inline`** functions. You can change this by using the `debug_inlines` pragma. Refer to *Pragmas controlling debugging* on page 3-4 for more information.

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

`-gt[`*`letters`*`]`

When used with `-g+`, 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 are generated. This is the default.

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

The effect of:

`-g+ -gt`*`T-options`*

can be combined in:

`-g`*`T-options`*

`-gx[`*`letters`*`]`

The following options are obsolete and are provided for backwards compatibility only. Use the `-O` options instead. Refer to *Defining optimization criteria* on page 2-25.

These options specify the level of optimization allowed when generating debug tables:

`-gx` No optimizations. This has the same effect as the `-O0` option.

`-gxr` Unoptimized register allocation. This option is supported in this release of the Software Development Toolkit, but its use is not recommended.

`-gxo` This option has the same effect as the `-O1` option.

Debug table format options

The following options control the format of the debug table generated when debug table generation is turned on with `-g+`:

`-asd` Use ASD debug table format. This option is obsolete and is provided for backwards compatibility only.

`-dwarf` Use DWARF1 debug table format. This option is obsolete and is provided for backwards compatibility only. Specify `-dwarf1` or `-dwarf2`.

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

If DWARF1 debug tables are generated and a procedure call standard that does not use a frame-pointer register is used (always the case with Thumb, and the default with ARM), local variables that have been allocated to the stack cannot be displayed by the debugger. In addition, stack backtrace is not possible.

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

2.3.9 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 3-2 for information on additional code generation options that are controlled using pragmas. This section describes:

- *Defining optimization criteria* on page 2-25
- *Setting the number of instructions for literals* on page 2-26

- *Controlling code and data areas* on page 2-26
- *Setting endianness* on page 2-28
- *Controlling SWI calling standards* on page 2-28
- *Setting load and store options* on page 2-28
- *Setting alignment options* on page 2-28.

Defining optimization criteria

`-Ospace` Optimize to reduce image size at the expense of increased execution time. For example, large structure copies are done by out-of-line function calls instead of inline code.

`-Otime` Optimize to reduce execution time at the expense of a larger image. For example, compile:

```
while (expression) body…;
```

as:

```
if (expression) {
    do body…;
    while (expression);
}
```

If neither `-Otime` or `-Ospace` is specified, the compiler uses a balance between the two. You can compile time-critical parts of your code with `-Otime`, and the rest with `-Ospace`. You should not specify both `-Otime` and `-Ospace` at the same time.

`-O`*`number`* Specifies the level of optimization to be used. The optimization levels are:

`-O0` Turn off all optimization, except some simple source transformations. This is the default optimization level if debug tables are generated with `-g+`. It gives the best debug view and the lowest level of optimization.

`-O1` Turn off the following optimizations:

- structure splitting
- range splitting
- cross-jumping
- conditional execution.

If this option is specified and debug tables are generated with `-g+` it gives a satisfactory debug view with good code density.

`-O2` Generate fully optimized code. If used with `-g+,` this option produces fully optimized code that is acceptable to the debugger, though the mapping of object code to source code is not always clear.

This is the default optimization level if debug tables are not generated.

Optimization options have the following effect on the debug view produced when compiling with `-g+`:

- If DWARF1 debug tables are generated (`-dwarf1` compiler option) and a procedure call standard that does not use a frame-pointer register is used (always the case with Thumb, and the default with ARM), local variables that have been allocated to the stack cannot be displayed by the debugger, regardless of optimization level. In addition, stack backtrace is not possible.
- For all debug tables formats, if optimization levels 1 and higher are used the debugger may display misleading values for local variables. If a variable is not live at the point where its value is interrogated, its location may be used for some other variable. In addition, the compiler replaces some local variables with constant values at each place the variable is used.
- For all debug table formats, if optimization level 2 is used the value of variables that have been subject to range splitting or structure splitting cannot be displayed.

Refer to *Pragmas controlling optimization* on page 3-4 for more information on controlling optimization.

Setting the number of instructions for literals

`-zi`*`Number`* The compiler selects a value for the maximum number of instructions allowed to generate an integer literal inline before using `LDR rx,=`*`value`* on the basis of the `-Otime`, `-Ospace`, and `-processor` options.

You can alter this behavior by setting *`Number`* to an integer between 1 and 4. Lower numbers generate less code at the possible expense of speed, depending on your memory system. The effect of altering this value is small, and is usually not significant.

Controlling code and data areas

`-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. This also stops the compiler reusing a multiply occurring string literal.

`-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. The sign of **`char`** is set by the last option specified that would normally affect it. For example, if you specify both the `-ansic` and `-zc` options and you want to make **`char`** signed in ANSI C mode, you must specify the `-zc` option *after* the `-ansic` option.

`-zo`

Generates one AOF area for each function. This can result in increased code size.

Normally the compiler generates one AOF function for each C compilation unit, and at *least* one AOF function for each C++ compilation unit.

This option enables the linker to remove unused functions when the `-remove` linker option is specified.

`-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. This may fault code that conforms to the ANSI specification.

This option has effect only for C, not for C++. Tentative declarations are not permitted in 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.

For example, you can force uninitialized global variables of any size to be placed in the ZI data area by specifying `-zz0`, though this may increase your code size.

Use this option in combination with `-zt` to avoid increased code size. 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 global 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. You must use this option when compiling code that contains inline SWIs.

Setting load and store options

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

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

`-za1` `LDR` may only access 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.

You can use this option to reduce interrupt latency. Note that the inline assembler is not subject to the limit imposed by the `-zr` option.

The Thumb compiler does not support this option.

Setting 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 ARM and Thumb compilers. 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.

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

`-zap1` Pointers to structures are assumed to be aligned on at least the minimum byte alignment boundaries set by `-zas`. This is the default.

`-zap0` Pointers to structures are not assumed to be aligned on at least the minimum byte alignment boundaries set by `-zas`. Casting `short[ ]` to `struct {short, short,...}` does not cause a problem.

`-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 the ARM compilers and 1 for the Thumb compilers.

2.3.10 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 2-32 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.

`-Wb` Suppresses the warning message:

`ANSI C forbids bit field type ’type’`

where `’type’` is **`char`** or **`short`**.

`-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`**, or where the target system defines 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 narrowing casts 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 `-fussy` 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 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++.

`-Wx` Disables *not used* warnings such as:

```
Warning: function ’foo’ declared but not used
```

2.3.11 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.

`-fh` Checks that:

- all external objects are declared before use
- all file-scoped static objects are used
- all predeclarations of static functions are used between their declaration and their definition. For example:

```
static int f(void);
static int f(void){return 1;}
line 2: Warning: unused earlier static declaration
of ’f’
```

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.

This option also causes casts to the same type to produce a warning. For example:

```
int f(int i) {return (int)i;}
             // Warning: explicit cast to same type.
```

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

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

2.3.12 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.

Caution

These options force the compiler to accept C and C++ source that would normally produce errors. If you use any of these options to ignore error messages, it means that your source code does not conform to the appropriate C or C++ standard.

These options may be useful during development, or when importing code from other environments, however they may allow code to be produced that does not function correctly and you are advised to correct the code, rather than using these options.

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` This option applies to C++ only. 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.

2.3.13 Pragma emulation

The following compiler option emulates `#pragma` directives:

`-zp[No_]`*`FeatureName`*

Emulates `#pragma` directives. See *Pragmas* on page 3-2 for details.

Chapter 3
ARM Compiler Reference

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

3.1 Compiler-specific features

This section describes the compiler-specific features supported by the ARM C and C++ compilers, including:

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

The features described here are not portable to other compilers.

3.1.1 Pragmas

Pragmas are recognized by the compiler in the following form:

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

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

```
-zp[no_]FeatureName
```

Table 3-1 lists the pragmas recognized by the ARM compilers.

Table 3-1 Pragmas

Pragma Name	On by default?
check_memory_accesses	No
check_printf_formats	Yes
check_scanf_formats	Yes
check_stack	Yes
continue_after_hash_error	No
debug	Yes
debug_inlines	No
force_top_level	No
include_only_once once (synonym)	No
optimize_crossjump	Yes
optimize_cse	Yes

Table 3-1 Pragmas (Continued)

Pragma Name	On by default?
optimize_multiple_loads	Yes
optimize_scheduling	Yes
side_effects	Yes

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 2-18 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. This affects only the file currently being processed.

`once` is a synonym for `include_only_once.`

`force_top_level`

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

Marks a function declared as a printf-like function, so that the arguments are type checked against the literal format string. If the format is not a literal string, no type checking is done. The format string must be the last fixed argument. For example:

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

`check_scanf_formats`

Marks a function declared as a scanf-like function, so that the arguments are type checked against the literal format string. If the format is not a literal string, no type checking is done. The format string must be the last fixed argument. For example:

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

Pragmas controlling debugging

The following pragmas control aspects of debug table generation:

`debug`
Controls whether debug tables are produced. If `#pragma no_debug` is specified, debug tables are not generated until the next `#pragma debug`. This pragma overrides the `debug_inlines` pragma. It has effect only if debug table generation is enabled with the `-g+` compiler option.

`debug_inlines`

Controls whether debug tables are produced for **`inline`** functions. This pragma has effect only if debug table generation is enabled with the `-g+` compiler option.

By default the ARM compilers do not generate debug data for **`inline`** functions when compiling for debug. Calls to **`inline`** functions are treated in the same way as if `-g+` were not in use.

If `#pragma debug_inlines` is specified, debug tables are generated in the following ways:

- If DWARF 1 debug tables are generated (`-dwarf1` compiler option), **`inline`** functions are treated as though they were declared **`static`**, rather than **`inline`**.
- If DWARF 2 debug tables are generated (the default), the compiler generates a common code AOF area for each **`inline`** function at its point of definition and never inlines calls to **`inline`** functions.

Pragmas controlling optimization

The following pragmas allow fine control over where specific optimizations are applied. Refer to *Defining optimization criteria* on page 2-25 for information on optimization options that are specified from the compiler command line.

Note

If you are turning off specific optimizations to stop the compiler from optimizing memory access instructions, consider using the **`volatile`** qualifier instead. **`volatile`** is also available in PCC mode unless `-fussy` is specified. Refer to *volatile* on page 3-13 for more information.

`optimize_crossjump`

Controls cross-jumping (the *common tail* optimization). This is disabled if the `-Otime` compiler option is specified.

`optimize_cse`

Controls common sub-expression elimination.

`optimize_multiple_loads`

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

`optimize_scheduling`

Controls instruction scheduling. Instruction scheduling is the re-ordering of machine instructions to suit the target processor. The ARM compilers perform instruction scheduling after register allocation and code generation.

If the `-processor StrongARM1` command-line option is specified, the compiler performs instruction scheduling for the StrongARM1. Otherwise, the compiler performs ARM9 instruction scheduling.

Instruction scheduling is on by default.

`no_side_effects`

Use of this pragma is deprecated. Use the `__pure` keyword instead. This pragma 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 enables you to tell the compiler that specific functions are candidates for common sub-expression elimination.

A function is properly defined as pure only if its result depends solely on the value of its scalar argument. This means that a pure function 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.

`no_side_effects` has the same effect as the `__pure` keyword.

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 2-24 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 should be used only if the function uses less than 256 bytes 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.

`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, halfword, and word writes respectively. It is up to your library implementation to check that the address given is reasonable.

3.1.2 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 compilers.

`__inline` This instructs the compiler to compile C functions inline. 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 compiles functions inline when `__inline` is used, except when `debug_inlines` is specified (see *Pragmas controlling debugging* on page 3-4). Code density and performance could be adversely affected if this keyword is specified for large functions that are called from multiple places.

`__irq` This enables a C or 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 keyword is not available in tcc or tcpp.

Refer to Chapter 9 *Handling Processor Exceptions* in the *ARM Software Development Toolkit User Guide* for detailed information on using `__irq`.

`__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 tells the compiler that specific functions are candidates for common sub-expression elimination.

A function should only properly defined as pure if its result depends only on the value of its arguments and has no side effects. This means that it 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 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_proc(int procnum);
```

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);
```

The `__value_in_regs` qualifier 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 defined in the ARM Procedure Call Standard.

Refer to Chapter 9 *Handling Processor Exceptions* in the *ARM Software Development Toolkit User Guide* for more information.

`__swi_indirect`

An indirect SWI that takes the number of the SWI to call as an argument in r12 can be described by a function declaration such as:

```
int __swi_indirect(ind_num) swi_name(int real_num,
                 int arg1, … argn);
```

where:

ind_num

Is the SWI number used in the SWI instruction.

real_num

Is the SWI number passed in r12 to the SWI handler. It can be specified at function call.

For example:

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

This can be called as:

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

It compiles to a SWI 0 with IOCTL+4 in r12.

Note that your system SWI handlers must support `__swi_indirect`.

`__value_in_regs`

This instructs the compiler to return a non-packed structure of up to four words 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);
```

Refer to the *ARM Software Development Toolkit User Guide* for information on the default method of passing and returning structures.

`__weak` If any **`extern`** function or object declaration is weak, the linker does not fault an unresolved reference to the function or object. If the reference remains unresolved, its value is assumed to be zero.

If the reference is made from code that compiles to a Branch or Branch Link instruction, the reference is resolved as branching to itself. It is not possible to reference an unresolved weak symbol with a Branch or Branch Link instruction only. There must be a guard, such as:

```
__weak void f(void);
...
if(f) f();
```

`___weak` (three underscores) is a synonym for `__weak` (two underscores).

3.1.3 Variable declaration keywords

This section describes the implementation of various standard and ARM-specific variable declaration keywords. Standard C or C++ keywords that do not have ARM-specific behavior or restrictions are not documented. See also *Type qualifiers* on page 3-11 for information on qualifiers such as **`volatile`**, and `__packed`.

Standard keywords

Note

The following keywords can be used only on local variables.

`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. The double precision floating-point type **`double`** occupies two ARM registers if the software floating-point library is used.

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

Note

The following keywords can be used only on global variables.

The following variable declaration keywords allow you to specify that a declared variable is allocated to a global register variable:

`__global_reg(n)`

Allocates the declared variable to a global integer register variable. Global register variables cannot be qualified or initialized at declaration. Valid types are:

- Any integer type except **`long long`**.
- Any pointer type.

`__global_freg(n)`

Allocates the declared variable to a global floating-point register variable. The variable must have type **`float`** or **`double`**. This keyword is legal only if `-fpu fpa` or `-apcs /hardfp` is specified.

The global register must be specified in all declarations of the same variable. For example, the following is an error:

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

In addition, `__global_reg` variables in C cannot be initialized at definition. For example, the following is an error in C, and not in C++:

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

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 **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 does not declare the global register variable, the function will read the wrong values from its supposed global register variables.

3.1.4 Type qualifiers

This section describes the implementation of various standard C, 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. Standard qualifiers that do not have ARM-specific behavior or restrictions are not documented.

Note

__packed is not, strictly speaking, a type qualifier. It is included in this section because it behaves like a type qualifier in most respects.

__packed

The __packed qualifier sets the alignment of any valid type to 1. This means:

- there is no padding inserted to align the packed object
- objects of packed type are read or written using unaligned accesses.

The __packed qualifier cannot be used on:

- floating-point types
- structures or unions with floating-point fields.

The __packed qualifier does not affect local variables of integral type.

The top level alignment specified with the `-zat` compiler option is applied to global objects. This means that there can be padding between two packed globals. The structure alignment specified with the `-zas` compiler option does not apply to packed structures.

Packed types must be specified with `__packed`. If you want to use `packed` rather than `__packed`, you must define it:

```
#define packed __packed
```

The `__packed` qualifier applies to all members of a structure or union when it is declared using `__packed`. There is no padding between members, or at the end of the structure. All sub-structures of a packed structure must be declared using `__packed`.

A packed structure or union is not assignment compatible with the corresponding unpacked structure. Because the structures have a different memory layout, the only way to assign a packed structure to an unpacked structure is by a field by field copy.

The effect of casting away `__packed` is undefined. The effect of casting a non-packed structure to `__packed` is undefined. It is legal to implicitly cast a pointer to an integral type to a pointer to a packed integral type.

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

Note

On ARM processors, access to unaligned data can be expensive, taking up to seven instructions and three work registers. Data accesses through packed structures should be minimized to avoid increase in code size, or performance loss.

The `__packed` qualifier is useful to map a structure to an external data structure, or for accessing unaligned data, but it is generally not useful to save data size because of the relatively high cost of access.

See Example 3-1 on page 3-13.

Example 3-1

```
typedef __packed struct
{
    char x;     // all fields inherit the __packed qualifier
    int y;
}X;             // 5 byte structure, natural alignment = 1

int f(X *p)
{
    return p->y;// does an unaligned read
}
typedef struct
{
    short x;
    char y;
    __packed int z;
    char a;
}Y;             // 8 byte structure, natural alignment = 2

int g(Y *p)
{
    return p->z + p->x; // only unaligned read for z
}
```

volatile

The standard qualifier **`volatile`** informs 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 and C++, a **`volatile`** qualified object is *accessed* if any word or byte (or halfword on ARM architecture that have halfword support) 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 ARM architectures that do not have halfword support.

3.2 C and C++ implementation details

This section describes implementation details for the ARM 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.

This section describes details for both C and C++ implementations. Where a feature applies specifically to one language, it is noted in the text.

3.2.1 Character sets and identifiers

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

- 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. An identifier may also contain a dollar `($)` character unless the `-fussy` compiler option is specified.
- 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.
- The ARM compilers do not support multibyte 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 and C++ libraries support 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 chars/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 false 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 false in any given retargeting of the generic ARM C library).
- No locale is used to convert multibyte characters into the corresponding wide characters (codes) for a wide character constant. This is not relevant to the generic implementation.

The character escape codes are shown in Table 3-2.

Table 3-2 Escape codes

Escape sequence	Char value	Description
`\a`	7	Attention (bell)
`\b`	8	Backspace
`\f`	9	Form feed
`\n`	10	New line
`\r`	11	Carriage return
`\t`	12	Tab

Table 3-2 Escape codes (Continued)

Escape sequence	Char value	Description
`\v`	`13`	Vertical tab
`\xnn`	`0xnn`	ASCII code in hexadecimal
`\nnn`	`0nnn`	ASCII code in octal

3.2.2 Basic data types

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

Size and alignment of basic data types

Table 3-3 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 `-zat2` is specified, a global **`char`** variable has an alignment of 2. This option is described in *Setting alignment options* on page 2-28.
- Local variables are always word aligned. For example, a local **`char`** variable has an alignment of 4.
- The natural alignment of a packed type is 1.

Table 3-3 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)

Table 3-3 Size and alignment of data types (Continued)

Type	Size in bits	Natural alignment
long double	64	4 (word aligned)
all pointers	32	4 (word aligned)
bool[a]	32	4 (word aligned)

[a] Applies to C++ only.

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. The high word of **long long** integers is always at the highest address. Refer to *Operations on integral types* on page 3-18 for more information.

Float

Floating-point quantities are stored in IEEE format. **float** values are represented by IEEE single precision values. **double** and **long double** values are represented by IEEE double precision values. 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. Refer to *Operations on floating-point types* on page 3-19 for more information.

Pointers

The following remarks apply to all pointer types in C, and to all pointer types except pointers to members in C++:

- 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`**.

Refer to *Pointer subtraction* on page 3-19 for more information.

3.2.3 Operations on basic data types

The ARM compilers perform the usual arithmetic conversions set out in relevant sections of the C and C++ standards. 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 by discarding an appropriate number of most significant bits. If the original number was too large, positive or negative, for the new type, there is no guarantee that the sign of the result will be the same as the original.
- 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 the destination type (**`int`** or **`long long`**).
- Floating-point underflow is disabled by default.
- Floating-point overflow raises a SIGFPE exception by default.
- Floating-point divide by zero raises a SIGFPE exception by default.

Pointer subtraction

The following remarks apply to all pointers in C, and to pointers other than pointers to members in C++:

- 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 one, two, or four bytes, the natural alignment of the object 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 appropriate C or C++ 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 if a, b, and c are integer expressions.

- 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.
- The compiler may evaluate function arguments in any order.

Any detail of order of evaluation not prescribed by the relevant standard may vary between releases of the ARM compilers.

3.2.4 Structured data types

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

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 data. The command-line option `-fy` sets the underlying type of **`enum`** to **`signed int`**. Refer to Chapter 2 *The ARM Compilers* for more information on the `-fy` option.

Structures

The following points apply to:

- all C structures
- all C++ structures and classes that do not have 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 *Setting alignment options* on page 2-28.

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 in packed structures* on page 3-24 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 3-1 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 on the stack or heap, such as those 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.

```
struct {char c; int x; short s;} example;
```

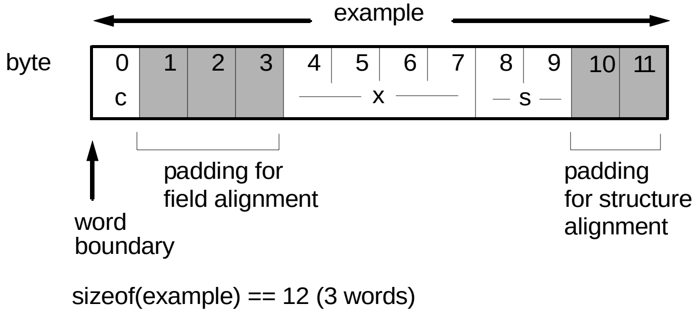


Figure 3-1 Conventional structure example

Bitfields

The ARM 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.

Note

The compiler warns about non **`int`** bitfields. You can disable this warning with the `-Wb` compiler option.

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`	6 bits
`y`	an 8-bit bitfield
`z`	a 5-bit bitfield
`unallocated`	3 unallocated bits.

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.

Bitfields in packed structures

Bitfield containers in packed structures have an alignment of 1. Therefore, the maximum bit padding for a bitfield in a packed structure is 7 bits. For an unpacked structure, the maximum padding is `8*sizeof(container-type)–1 bits`.

Packed structures

A packed structure is one in which the alignment of the structure, and of the fields within it, is always 1, independent of the alignment specified by the `-zas` compiler option. Floating-point types cannot be fields of packed structures.

Packed structures are defined with the `__packed` qualifier. There is no command-line option to change the default packing of structures. Refer to *Type qualifiers* on page 3-11 for more information.

3.3 Standard C implementation definition

Appendix G of the ISO C standard (IS/IEC 9899:1990 (E)) collects together information about portability issues. Subclause G3 lists the behavior that each implementation must document.

The following subsections correspond to the relevant sections of subclause G3. It describes aspects of the ARM C compiler and ANSI C library that are not defined by the ISO C standard, and that are implementation-defined.

Note

This section does not duplicate information that is applicable to both C and C++ implementations. This is provided in *C and C++ implementation details* on page 3-14. This section provides references where applicable.

3.3.1 Translation

Diagnostic messages produced by the compiler are of the form:

`source-file``,` *`line-number`*`:` *`severity`*`:` *`explanation`*

where *`severity`* is one of:

`Warning`	A helpful message from the compiler.
`Error`	A violation of the ANSI specification from which the compiler is able to recover by guessing the intention.
`Serious error`	A violation of the ANSI specification from which no recovery is possible because the intention is not clear.
`Fatal`	An indication that the compiler limits have been exceeded, or that the compiler has detected a fault in itself (for example, not enough memory).

3.3.2 Environment

The mapping of a command line from the ARM-based environment into arguments to `main()` is implementation-specific. The generic ARM C library supports the following:

- main
- interactive device
- standard input, output, and error streams.

main()

The arguments given to `main()` are the words of the command line (not including I/O redirections), delimited by white space, except where the white space is contained in double quotes.

Note that:

- a whitespace character is any character of which `isspace()` is true
- a double quote or backslash character (\) inside double quotes must be preceded by a backslash character
- an I/O redirection will not be recognized inside double quotes.

Interactive device

In an unhosted implementation of the ARM C library, the term *interactive device* may be meaningless. The generic ARM C library supports a pair of devices, both called `:tt`, intended to handle a keyboard and a VDU screen. In the generic implementation:

- no buffering is done on any stream connected to `:tt` unless I/O redirection has taken place
- if I/O redirection other than to `:tt` has taken place, full file buffering is used (except where both `stdout` and `stderr` have been redirected to the same file, where line buffering is used).

Standard input, output and error streams

Using the generic ARM C library, the standard input, output and error streams `stdin`, `stdout`, and `stderr` can be redirected at runtime. For example, if `mycopy` is a program which simply copies the standard input to the standard output, the following line runs the program:

```
mycopy < infile > outfile 2> errfile
```

and redirects the files as follows:

`stdin`	is redirected to `infile`
`stdout`	is redirected to `outfile`
`stderr`	is redirected to `errfile`

The following shows the permitted redirections:

`0<` *`filename`*	reads `stdin` from *`filename`*
`<` *`filename`*	reads `stdin` from *`filename`*
`1>` *`filename`*	writes `stdout` to *`filename`*
`>` *`filename`*	writes `stdout` to *`filename`*
`2>` *`filename`*	writes `stderr` to *`filename`*
`2>&1`	writes `stderr` to the same place as `stdout`
`>&` *`filename`*	writes both `stdout` and `stderr` to *`filename`*
`>>` *`filename`*	appends `stdout` to *`filename`*
`>>&` *`filename`*	appends both `stdout` and `stderr` to *`filename`*

3.3.3 Identifiers

Refer to *Character sets and identifiers* on page 3-14 for details.

3.3.4 Characters

Refer to *Character sets and identifiers* on page 3-14 for a description of the characters in the source character set.

3.3.5 Integers

Refer to *Integer* on page 3-17 for details.

3.3.6 Floating-point

Refer to *Float* on page 3-17 for details. In addition, floating-point support is documented in Chapter 11 *Floating-point Support*.

3.3.7 Arrays and pointers

Refer to *Pointers* on page 3-17 for details.

3.3.8 Registers

Using the ARM C compiler, you can declare any number of objects to have the storage class **`register`**.

Refer to *Variable declaration keywords* on page 3-9 for a description of how ARM implement the **`register`** storage class.

3.3.9 Structures, unions, enumerations, and bitfields

The ISO/IEC C standard requires that the following implementation details are documented for structured data types:

- The outcome when a member of a union is accessed using a member of different type.
- The padding and alignment of members of structures.
- Whether a plain **`int`** bitfield is treated as a **`signed int`** bitfield or as an **`unsigned int`** bitfield.
- The order of allocation of bitfields within a unit.
- Whether a bitfield can straddle a storage-unit boundary.
- The integer type chosen to represent the values of an enumeration type.

These implementation details are documented in the relevant sections of *C and C++ implementation details*.

Unions

Refer to *Unions* on page 3-20 for details.

Padding and alignment of structure members

Refer to *Structures* on page 3-20 for details.

Enumerations

Refer to *Enumerations* on page 3-20 for details.

Bitfields

Refer to *Bitfields* on page 3-22 for details.

3.3.10 Qualifiers

An object that has volatile-qualified type is *accessed* if any word or byte (or halfword on ARM architectures that have halfword support) of it 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-qualified **`short`** is undefined on ARM architectures that do not have halfword support.

3.3.11 Declarators

The number of declarators that may modify an arithmetic, structure or union type is limited only by available memory.

3.3.12 Statements

The number of case values in a **`switch`** statement is limited only by memory.

3.3.13 Preprocessing directives

A single-character constant in a preprocessor directive cannot have a negative value.

The ANSI standard header files are contained within the compiler itself and may be referred to in the way described in the standard (using, for example, `#include <stdio.h>`, etc.).

Quoted names for includable source files are supported. The compiler will accept host filenames or UNIX filenames. In the latter case, on non-UNIX hosts, the compiler tries to translate the filename to a local equivalent.

The recognized `#pragma` directives are shown in *Pragmas* on page 3-2.

The date and time of translation are always available, so `__DATE__` and `__TIME__` always give the date and time respectively.

3.3.14 Library functions

The precise attributes of a C library are specific to a particular implementation of it. The generic ARM C library has or supports the following features:

- The macro `NULL` expands to the integer constant 0.
- If a program redefines a reserved external identifier, an error may occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error will be detected.

- The `assert()` function prints the following message and then calls the `abort()` function:

  ```
  *** assertion failed: expression, file filename, line
  linenumber
  ```

The following functions usually test only for characters whose values are in the range 0 to 127 (inclusive):

- `isalnum()`
- `isalpha()`
- `iscntrl()`
- `islower()`
- `isprint()`
- `isupper()`
- `ispunct()`

Characters with values greater than 127 return a result of 0 for all these functions except `iscntrl()` which returns non-zero for 0 to 31, and 128 to 255.

Setlocale call

After the call `setlocale(LC_CTYPE, "ISO8859-1"),` the statements in Table 3-4 apply to character codes and affect the results returned by the `ctype()` functions:

Table 3-4 Character codes

Code	Description
128 to 159	control characters
160 to 191	punctuation
192 to 214	uppercase
215	punctuation
216 to 223	uppercase
224 to 246	lowercase
247	punctuation
248 to 255	lowercase

Mathematical functions

The mathematical functions return the values in Table 3-5.

Table 3-5 Mathematical functions

Function	Condition	Returned value
`log(x)`	x <= 0	`–HUGE_VAL`
`log10(x)`	x <= 0	`–HUGE_VAL`
`sqrt(x)`	x < 0	`–HUGE_VAL`
`atan2(x,y)`	x = y = 0	`–HUGE_VAL`
`asin(x)`	abs(x) > 1	`–HUGE_VAL`
`acos(x)`	abs(x) > 1	`–HUGE_VAL`
`pow(x,y)`	x=y=0	`–HUGE_VAL`

Where `–HUGE_VAL` is returned, a number is returned which is defined in the header `math.h`. Consult the `errno` variable for the error number.

The mathematical functions set `errno` to `ERANGE` on underflow range errors.

A domain error occurs if the second argument of `fmod()` is zero, and `–HUGE_VAL` is returned.

Signal function

The set of signals for the generic `signal()` function shown in Table 3-6.

Table 3-6 Signal function signals

Signal	Description
SIGABRT	abort
SIGFPE	arithmetic exception
SIGILL	illegal instruction
SIGINT	attention request from user

Table 3-6 Signal function signals (Continued)

Signal	Description
SIGSEGV	bad memory access
SIGTERM	termination request
SIGSTAK	stack overflow

The default handling of all recognized signals is to print a diagnostic message and call `exit()`. This default behavior applies at program startup.

When a signal occurs, if `func` points to a function, the equivalent of `signal(sig, SIG_DFL)` is first executed. If the **SIGILL** signal is received by a handler specified to the `signal()` function, the default handling is reset.

Generic ARM C library

The generic ARM C library also has the following characteristics relating to I/O. Note that a given targeting of the ARM C library may not have the same characteristics:

- The last line of a text stream does not require a terminating newline character.
- Space characters written out to a text stream immediately before a newline character do appear when read back in.
- No null characters are appended to a binary output stream.
- The file position indicator of an append mode stream is initially placed at the end of the file.
- A write to a text stream does not cause the associated file to be truncated beyond that point (device dependent).
- The characteristics of file buffering are as intended by section 4.9.3 of the ANSI C standard. The maximum number of open files is set in `stdio.h` as:

  ```
  #define _SYS_OPEN 16
  ```

  if Angel is in use.
- A zero-length file (in which no characters have been written by an output stream) does exist.
- The same file can be opened many times for reading, but only once for writing or updating. A file cannot be open simultaneously for reading on one stream and for writing or updating on another.

- Local time zones and Daylight Saving Time are not implemented. The values returned will always indicate that the information is not available.
- The status returned by `exit()` is the same value that was passed to it. For definitions of `EXIT_SUCCESS` and `EXIT_FAILURE`, refer to the header file `stdlib.h`.
- The error messages returned by the `strerror()` function are identical to those given by the `perror()` function.
- If the size of area requested is zero, `calloc()`, `malloc()` and `realloc()` return `NULL`.
- `abort()` closes all open files, and deletes all temporary files.
- `fprintf()` prints `%p` arguments in hexadecimal format (lowercase) as if a precision of 8 had been specified. If the variant form (`%#p`) is used, the number is preceded by the character `@`.
- `fscanf()` treats `%p` arguments identically to `%x` arguments.
- `fscanf()` always treats the character "-" in a `%...[...]` argument as a literal character.
- `ftell()` and `fgetpos()` set `errno` to the value of EDOM on failure.
- `perror()` generates the messages in Table 3-7.

Table 3-7 perror() messages

Error	Message
`0`	`No error (errno = 0)`
`EDOM`	`EDOM - function argument out of range`
`ERANGE`	`ERANGE - function result not representable`
`ESIGNUM`	`ESIGNUM - illegal signal number to signal() or raise()`
others	`Error code number has no associated message`

The following characteristics, required to be specified in an ANSI-compliant implementation, are unspecified in the generic ARM C library:

- the validity of a filename
- whether `remove()` can remove an open file
- the effect of calling the `rename()` function when the new name already exists
- the effect of calling `getenv()` (the default is to return `NULL`, no value available)
- the effect of calling `system()`
- the value returned by `clock().`

3.4 Standard C++ implementation definition

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

When used in ANSI C mode, the ARM C++ compilers are identical to the ARM C compiler. Refer to *Standard C implementation definition* on page 3-25 for details.

3.4.1 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 bitfield width). Otherwise, the value is truncated to fit the size of the destination type and a warning is given.

3.4.2 Standard C++ library implementation definition

This section describes the Rogue Wave Standard C++ library that is supplied with ARM C++. 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 SDT installation directory.

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 from 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 *Library support* on page 2-5.

The library is also supplied as pre-built object files to enable you to rebuild armcpplib if required. Refer to Chapter 4 The C and C++ Libraries for more information on rebuilding armcpplib.

Table 3-8 on page 3-36 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 *Library support* on page 2-5 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 3-8 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 24.5.3 and 24.5.4)
25	Algorithms library
26.2	Complex number
26.4	Generalized numeric operations

3.5 C and C++ language extensions

This section describes the language extensions supported by the ARM C and C++ compilers.

3.5.1 C Language Extensions

The compilers support the following extensions to the ANSI C language. None of these extensions is available if the compiler is restricted to compiling strict ANSI C, for example, by specifying the `-fussy` compiler option.

// comments

The character sequence `//` starts a comment. As in C++, the comment is terminated by the next newline character. Note that comment removal takes place after line continuation has been performed, so:

```
// this is a - \
single comment
```

The characters of a comment are examined only to find the comment terminator, therefore:

- `//` has no special significance inside a comment introduced by `/*`
- `/*` has no special significance inside a comment introduced by `//`

3.5.2 C and C++ language extensions

The compilers support the following extensions to both the ANSI C language, and the Draft ISO/IEC C++ language. Refer to *C Language Extensions* for language extensions that apply only to C. None of these extensions is available if the compiler is restricted to compiling strict ANSI C or strict draft C++, for example by specifying the `-fussy` compiler option.

Identifiers

The $ character is a legal character in identifiers.

Void returns and arguments

Any **void** type is permitted as the return type in a function declaration, or the indicator that a function takes no argument. For example, the following is permitted:

```
typedef void VOID;
int fn(VOID);       // Error in -fussy C and C++
VOID fn(int x);     // Error in -fussy C
```

Long long

ARM C and C++ support 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**. This means that the value of the expression 2147483648 > –1 is 0 in strict C and C++, and 1 in ARM C and C++.

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

- **long long** bitfields are not supported.
- **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**.

Inline Assembler

The ARM C compilers support inline assembly language with the __asm specifier.

The ARM C++ compilers support the syntax proposed in the Draft C++ Standard, with the restriction that the string-literal must 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 C compiler `__asm` syntax when used with both **asm** and `__asm`.

The ARM inline assembler implements the full ARM instruction set, including generic coprocessors, halfword instructions and long multiply.

The Thumb inline assembler implements the full Thumb instruction set.

Syntax

The inline assembler is invoked with the assembler specifier, and is followed by a list of assembler instructions inside braces. For example:

```
__asm
{
    instruction [; instruction]
    ...
    [instruction]
}
```

If two instructions are on the same line, you must separate them with a semicolon. If an instruction is on multiple lines, line continuation must be specified with the backslash character (\). C or C++ comments may be used anywhere within an inline assembly language block.

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

Refer to Chapter 8 Mixed Language Programming in the *ARM Software Development Toolkit User Guide* for detailed information on using the inline assemblers from both C and C++.

3.6 Predefined macros

Table 3-9 lists the macro names predefined by the ARM C and 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 3-9 Predefined macros

Name	Value	Notes
`__STDC__`	1	defined in all compiler modes except PCC mode
`__cplusplus`	1	defined in C++ 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_NOSWST`		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)
`__APCS_FPREGARGS`		defined if `-apcs /fpregargs` is in use
`__BIG_ENDIAN`		defined if compiling for a big-endian target
`__DIALECT_FUSSY`		defined if `-fussy` is specified.
`__DIALECT_PCC`		defined if `-pcc` is specified.
`__TARGET_ARCH_`*`xx`*		*xx* represents the target architecture. The value of *xx* depends on the target architecture. For example, if the compiler options `-arch 4T` or `-cpu ARM7TDMI` are specified then `__TARGET_ARCH_4T` is defined, and no other symbol starting with `_TARGET_ARCH_` is defined.

Table 3-9 Predefined macros (Continued)

Name	Value	Notes
`__TARGET_CPU_`*xx*		*xx* represents the target cpu. The value of *xx* is derived from the `-processor` compiler option, or the default if none is specified. For example, if the compiler option `-processor 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. If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, `-processor SA-110` is mapped to `__TARGET_CPU_SA_110`.
`__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.
`__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
`__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 full pathname of the current source file
`__MODULE__`		contains the filename part of the value of `__FILE__`

Table 3-9 Predefined macros (Continued)

Name	Value	Notes
`__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

3.7 Implementation limits

This section lists implementation limits for the ARM C and C++ compilers.

3.7.1 Draft Standard Limits

The Draft C++ Standard standard recommends certain minimum limits that a conforming compiler should accept. You should be aware of these when porting applications between compilers. A summary is given in Table 3-10. A limit of `mem` indicates that no limit is imposed by the ARM compilers, other than that imposed by the availability of memory.

Table 3-10 Implementation limits

Description	Recommended	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.	65536	mem
Characters in a character string literal or wide string literal after concatenation.	65536	mem

Table 3-10 Implementation limits (Continued)

Description	Recommended	ARM
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 in a 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
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

3.7.2 Internal limits

In addition to the limits described in Table 3-10 on page 3-43, the compiler has internal limits listed in Table 3-11.

Table 3-11 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

3.8 Limits for integral numbers

The following table gives the ranges for integral numbers as implemented in ARM C and 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 by the ARM compilers.

When entering constants, you must be careful about the size and sign of the quantity. Constants are interpreted differently in decimal and hexadecimal/octal. See the appropriate C or C++ standard, or any of the recommended textbooks on the C and C++ programming language for more details.

Table 3-12 Integer ranges

Constant	Meaning	End-point	Hex Value
`CHAR_MAX`	Maximum value of **`cha`**r	`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`**	`–2147483648`	`0x80000000`
`LONG_MAX`	Maximum value of **`long`**	`2147483647`	`0x7fffffff`
`LONG_MIN`	Minimum value of **`long`**	`–2147483648`	`0x80000000`
`ULONG_MAX`	Maximum value of **`unsigned long`**	`4294967295`	`0xffffffff`

3.9 Limits for floating-point numbers

The following tables give the characteristics, ranges, and limits for floating-point numbers as implemented in ARM C and 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 3-13 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

Table 3-13 Floating-point limits (Continued)

Constant	Meaning	Value
`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 3-14 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

3.10 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++.

3.10.1 Major language feature support

Table 3-15 shows the major language features supported by this release of ARM C++.

Table 3-15 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	None
Libraries	17 to 27	Partial. Refer to *Standard C++ library implementation definition* on page 3-35.

3.10.2 Minor language feature support

Table 3-16 shows the minor language features supported by this release of ARM C++.

Table 3-16 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	Yes
Nothrow **`new`**	No, but **`new`** does not **`throw`**.
`bool` type	Yes

Table 3-16 Minor language feature support (Continued)

Minor Language Feature	Supported
`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

Chapter 4
The C and C++ Libraries

This chapter describes how to rebuild the ARM C and C++ libraries. It also describes how to retarget the standard C library to your own hardware and operating system environment. It contains the following sections:

4.1 About the runtime libraries

There are three runtime libraries provided to support compiled C and C++:

- the ANSI C library
- the C++ library
- the embedded C library.

Note

The C++ library is supplied with the ARM C++ compilers. The C++ compilers and C++ libraries are available separately. Contact your distributor, or ARM Limited if you want to purchase ARM C++.

The embedded and ANSI C libraries are supplied in:

- source form, for retargeting to your ARM-based hardware
- binary form, targeted at the common environment supported by the ARMulator, Angel, EmbeddedICE, and Multi-ICE, so that you can immediately run and debug programs under the ARMulator, or on a development board.

The Rogue Wave parts of the C++ libraries are supplied in binary form only, both as precompiled library variants and as sublibraries that allow you to rebuild the libraries. The ARM part of the C++ library is supplied as both source and binary.

Retargeting the ANSI C library requires some knowledge of ARM assembly language, and some understanding of the ARM processor and hardware being used. Refer to the following documentation for more information:

- the datasheet for your processor
- the assembly language chapters of the *ARM Software Development Toolkit User Guide* and the *ARM Software Development Toolkit Reference Guide*.
- the *ARM Architectural Reference Manual*.

4.1.1 The ANSI C library

The ARM C library conforms to the ANSI C library specification. The library is targeted at the common operating environment supported by ARM debugging software and hardware. Angel semihosting SWIs are called for C library functions, such as `printf()`, that require operating system support. This environment is supported by:

- the ARMulator, through its support for Angel SWIs
- the ARM evaluation and development boards, because these are supplied with Angel built into ROM
- EmbeddedICE and Multi-ICE.

Refer to Chapter 13 *Angel* in the *ARM Software Development Toolkit User Guide* for more information on Angel C library support and the Angel semihosting SWIs.

The ANSI C library contains:

- target-independent modules written in ANSI C, such as `printf()`
- target-independent modules written in ARM assembly language, such as `divide()` and `memcpy()`
- target-dependent modules written in ANSI C, such as default signal handlers like the `clock` module
- target-dependent modules written in ARM assembly language.

You can build the target-independent portions of the library immediately. If you are building a new APCS library variant you must modify the appropriate library makefile. If you are retargeting the C library to your own operating environment, you must modify the target-dependent portions of the library to implement them on your system.

Refer to *The ANSI C library* on page 4-5 for detailed information on the C library.

4.1.2 The C++ library

The ARM C++ library supports most of the standard library functions defined in the January 1996 C++ Draft Standard. ARM C++ library support consists of:

The ANSI C library

The ARM C++ compilers use the same ANSI C library as the ARM C compilers.

The ARM C++ library

This library consists of:

- The Rogue Wave Standard C++ library version 1.2.1.
- Helper functions for the C++ compiler, and a number of additional C++ functions not supported by the Rogue Wave library.

By default the C++ libraries are installed in `c:\ARM250\lib`. The accompanying header files are installed in `c:\ARM250\include`. Refer to *The ARM C++ libraries* on page 4-15 for detailed information on C++ library support.

4.1.3 The embedded C library

The embedded C library is a subset of the full ANSI C library that:

- does not use static data
- does not rely on underlying system functionality
- minimizes dependencies between functions within the library.

You can link the embedded C library with an application running from read-only memory on a target board. Refer to *The embedded C library* on page 4-19 for more information. Refer to Chapter 10 *Writing Code for ROM* in the *ARM Software Development Toolkit User Guide* for an example of how to use the embedded C library.

4.1.4 Library naming conventions

The ANSI C and C++ libraries are supplied as 26 precompiled variants targeted at Angel. The embedded C library is supplied as four precompiled variants. The library filenames are postfixed with letters and digits that identify the variant.

Library names have the form:

`libname_<apcs_variant>.<bits><bytesex>`

where:

`apcs_variant` identifies the APCS options with which the library was compiled. Refer to Table 4-1 on page 4-5 and Table 4-2 on page 4-6 for a list of precompiled C library variants supplied with this release. Refer to *Automatic inclusion of libraries* on page 6-39 for a full description of library naming conventions.

`bits` can be one of:

`32`	compiled as an ARM library.
`16`	compiled as a Thumb library.

`bytesex` can be one of:

`l`	compiled for a little-endian target.
`b`	compiled for a big-endian target.

4.2 The ANSI C library

This section describes the ANSI C library. It provides information on using the precompiled libraries, and retargeting the library.

4.2.1 Using the ANSI C library

The ANSI C library variants are listed in Table 4-1, and Table 4-2 on page 4-6.

If you are building your C code with APCS options that are supported by one of the precompiled library variants, the linker selects a precompiled library version that is compatible.

If you select APCS options for which a suitable precompiled library variant does not exist, the linker warns that the library file cannot be found. In this case you must build a library variant that supports your APCS options, and name it appropriately so that the linker selects your library variant.

Refer to *Retargeting the ANSI C library* on page 4-6 for more information on building a new library variant. Refer to *Automatic inclusion of libraries* on page 6-39 for more information on library selection and naming.

Table 4-1 Precompiled Thumb library variants

Thumb libraries	Description
`armlib.16l` `armlib.16b`	Compiled with no software stack checking.
`armlib_i.16l` `armlib_i.16b`	Compiled for interworking.
`armlib_s.16l` `armlib_s.16b`	Compiled with software stack checking.
`armlib_si.16l` `armlib_si.16b`	Compiled for interworking, and with software stack checking.

Table 4-2 Precompiled ARM library variants

Library	Description
`armlib.32l` `armlib.32b`	Compiled with software stack checking, frame pointer, and software floating-point.
`armlib_c.32l` `armlib_c.32b`	Compiled with no software stack checking.
`armlib_cn.32l` `armlib_cn.32b`	Compiled with no software stack checking and no frame pointer.
`armlib_h.32l` `armlib_h.32b`	Compiled with hardware floating-point, software stack checking, and frame pointer.
`armlib_hc.32l` `armlib_hc.32b`	Compiled with hardware floating-point and frame pointer, and with no software stack checking.
`armlib_hcn.32l` `armlib_hcn.32b`	Compiled with hardware floating-point, no software stack checking, and no frame pointer.
`armlib_r.32l` `armlib_r.32b`	Compiled with software stack checking, frame pointer, hardware floating-point, and fp arguments in fp registers.
`armlib_rc.32l` `armlib_rc.32b`	Compiled with frame pointer, hardware floating-point, fp arguments in fp registers, and no software stack checking.
`armlib_rcn.32l` `armlib_rcn.32b`	Compiled with hardware floating-point, fp arguments in fp registers, no software stack checking, and no frame pointer.

4.2.2 Retargeting the ANSI C library

The ANSI C library is supplied in full source form so that you can rebuild it, or retarget it to your own operating system and hardware environment.

The C library source files are installed in `c:\ARM250\cl`. The top level of this directory contains the generic assembly language and C source code.

Additional code is contained in the subdirectories listed in Table 4-3.

Table 4-3 C library subdirectories

`\angel`	Contains targeting code for the Angel *semihosted* C library. The library is called semihosted because many functions, such as file I/O, are implemented on the host computer, through the host C library.
`\embedded`	Contains embedded C library specific functions.
`\thumb`	Contains Thumb-specific implementations of C library functions.
`\clib.b`	The build directory. This directory contains a separate subdirectory for each standard variant of the C libraries.
`\fplib`	Contains source code for the software floating-point library.
`\fpehead`	Contains assembly language macros and definitions used by the software floating-point library code.

The target-independent code is generally grouped into one file for each section of the ANSI C library. Conditional compilation and assembly is used to build a fine-grain library, usually with one object file for each function.

The procedure for retargeting the C library is:

1. Copy the complete C library to the target directory and modify the source.
2. Define the Angel environment, if required.
3. Modify the makefile that matches your target environment the closest.
4. Make the library.

These steps are described in detail in the following sections.

Copying and modifying the sources

Follow these steps to copy and modify the C library source files:

1. Copy the complete `c:\ARM250\cl` source directory and all its contents to a new directory. The new directory is your target directory. If you copy the `\cl` directory outside `c:\ARM250` you must also copy `c:\ARM250\include` to the same directory.

2. Modify the source files as required.

   If you are rebuilding the existing ARM library with your own APCS options you can skip this step. Refer to *Editing the makefile* on page 4-8 for information on specifying options to build a new C library variant.

If you are porting the C library to your own hardware and operating system environment, you must provide code to implement the C library functions that are dependent on the target environment.

The target-dependent functions that you may need to implement are described in:

- *Target-dependent ANSI C library functions* on page 4-27
- *Target-dependent I/O support functions* on page 4-30
- *Target-dependent kernel functions* on page 4-36
- *Target-dependent operating system functions* on page 4-41.

In addition, if you are retargeting the library to your own operating environment you must supply code to initialize the runtime libraries. See *Initializing the runtime libraries* on page 4-10 for more information.

Defining the Angel environment

If you are building a library that uses Angel semihosting SWIs to access system function, you must supply the definition of the Angel environment. The ARM ANSI C library Angel definitions are in `c:\ARM250\cl\angel\h_os.h` and `c:\ARM250\cl\angel\h_os.s`.

These files contain C and assembly language definitions to support Angel SWIs and breakpoints.

Editing the makefile

The makefiles for all prebuilt library variants are installed in `c:\ARM250\cl\clib.b\`*`library_variant`* where *`library_variant`* is either:

- `angel_`*`apcs_variant`* for Angel-targeted libraries
- `embedded_`*`apcs_variant`* for embedded libraries.

The makefiles are named:

- `makefile.pc` for Windows makefiles
- `makefile.unix` for UNIX makefiles.

Follow these steps to edit the makefile:

1. Select the makefile that corresponds most closely to your target environment. For example, to build a C library that is targeted to a processor without a floating-point coprocessor, use a makefile that builds a library with `-apcs /softfp`.

2. Modify the makefile. The makefile has two parts:

- Definitions of selected options for the target environment. Specify the options you want in this part of the makefile.
- A list of object files that are built by the makefile. If you add new source files to the C library, you must include them in this list.

Refer to *Makefile options* on page 4-11 for a description of the makefile options.

3. Rename or copy the makefile to `makefile`. For example, from a Windows DOS command-line type:

```
copy makefile.pc makefile
```

Building the target-specific library

Follow these steps to build the target-specific library:

1. Change directory to the build subdirectory of the target directory containing the makefile. For example, from a Windows DOS command-line type:

   ```
   cd c:\ARM250\Cl\clib.b\angel_cn.32l
   ```

   to change to the build directory for a no software stack check, no frame pointer, little-endian library targeted at Angel.

2. From the DOS command-line type `armmake clean`

   to remove any object and library files from previous builds.

3. This step is required only if you are building from a Windows DOS command-line.

   Type `armmake via`

   This command constructs the via files that are used during the build to control assembly and compilation operations. The via files are required because of the command-line length restrictions imposed by MS-DOS.

4. Type `armmake all`

   This command makes your C library. The library is saved in the target directory, with the name defined in the LIBNAME makefile option.

Initializing the runtime libraries

Under Angel, the C runtime libraries are initialized as part of Angel initialization. If you are retargeting the C library to your own environment, your operating system or application must initialize the runtime libraries.

To initialize the libraries you must call `__rt_lib_init` with four parameters:

- the top of the stack
- the start of the code area (`Image$$RO$$Base`)
- the end of the code area (`Image$$RO$$Limit`)
- a pointer to a C++ init block `(__cpp_initialise` and `__cpp_finalise`).

Example 4-1 shows an implementation of the initialization code.

Example 4-1

```
    IMPORT  __rt_lib_init
    IMPORT  |Image$$RO$$Base|
    IMPORT  |Image$$RO$$Limit|
    IMPORT  |__cpp_initialise|, WEAK
    IMPORT  |__cpp_finalise|, WEAK

    AREA    LibInit, CODE, READONLY

init_library
    ADR     r0, TOS
    LDMIA   r0, {r0-r3}
    B       __rt_lib_init
    ; Return to the caller when we return...

TOS DCD     0x3fffffc               ; My Top Of Stack Address
    DCD     |Image$$RO$$Base|
    DCD     |Image$$RO$$Limit|
    DCD     |__cpp_initialise|
    DCD     |__cpp_finalise|

    END
```

Makefile options

The supplied makefiles for the C library variants have the following options:

LIBTYPE Specify the type of library you wish to build:

`=angel` build an Angel semihosting library.

`=embedded` build the embedded library.

LIBNAME Specify the name for your output library file. You must use a name with the correct APCS file extension if you want the linker to automatically link with your library. You can do this by specifying a name of the form:

`libname.$(PCSZ)$(ENDIAN)`

Refer to *Automatic inclusion of libraries* on page 6-39 for more information on naming conventions.

TARGET Specify whether the target system is ARM or Thumb:

`=arm` the target is ARM.

`=thumb` the target is Thumb.

memcpy Specify how the following memory functions are implemented:

`=small` The functions `memcpy()`, `memmove()`, and `memset()` are implemented by generic C code. The C code attempts to do as much work as possible in word units. Each function occupies approximately 100 bytes.

`=fast` The functions `memmove()` and `memcpy()` are implemented together in assembly language. The code attempts to do the move eight words at a time, using `LDM` and `STM` instructions.

The two functions occupy approximately 1200 bytes. The `memset()` function is implemented similarly, and occupies approximately 200 bytes.

divide Specify how divide is implemented:

`=small` A small but slow implementation of division (approximately one bit per iteration).

`=unrolled` Unsigned and signed divide are unrolled eight times for greater speed. This option increases code size. Complete unrolling of divide is possible, but should be done with care because the size increase might result in decreased performance on a cached ARM.

Both variants include fast unsigned and signed divide by 10.

stdfile_redirection

Specify whether command-line redirection is enabled:

`=on` Redirection is enabled. Standard UNIX-style file redirection syntax can be used from the image argument string (<, >, >>, `>&`, `1>&2`).

`=off` Redirection is disabled.

backtrace Specify whether stack backtracing is enabled. This option is obsolete and is provided for backwards compatibility only. Backtrace works only for code compiled to use a frame pointer:

`=on` The default signal handler ends by producing a call-stack traceback to `stderr`.

Use of this variant is not encouraged. It increases the proportion of the library that is linked into all images, and provides functionality that is better obtained from a separate debugger.

`=off` No backtracing selected.

stack Specify the type of stack used by the library:

`=contiguous` Use a contiguous stack.

`=chunked` Use a chunked stack.

fp_type Specify floating-point support:

`=fplib` Include software floating-point routines in the C library. Refer to Chapter 11 *Floating-point Support* for information on floating-point support.

`=hardfp` The library expects hardware floating-point support.

PCSZ Specify whether the library is built for 32-bit or 26-bit ARM processors:

`=32` Build for a 32-bit address space.

`=26` Build for a 26-bit address space. This option is obsolete and is provided for backwards compatibility only.

ENDIAN Select the endianness of the library:

`=b` Big-endian.

`=l` Little-endian.

FPIS Select the floating-point instruction set:

`=2` Release 2 FPA instructions. This option is obsolete and is provided for backwards compatibility only.

`=3` Release 3 FPA instructions.

FPREGARGS

Specify whether or not floating-point arguments are passed in floating-point registers:

`fpregargs` Floating-point arguments are passed in floating-point registers.

`nofpregargs` Floating-point arguments are passed in integer registers.

FPTYPE Select the floating-point APCS options. Set FPTYPE to either:

`softfp` Floating-point instructions are handled by software.

`hardfp/fpe$(FPIS)/$(FPREGARGS)`

Floating-point instructions are handled by hardware.

LDMLIMIT

Specify the maximum number of registers allowed in load multiple and store multiple instructions. You can use this option to minimize interrupt latency.

INTERWORK

Specify whether the library supports interworking between ARM code and Thumb code:

`=interwork` Build an interworking library variant.

`=nointerwork` Build a non-interworking library variant.

APCSA Specify additional APCS options for the assembler. Refer to Chapter 5 *Assembler* for details of the APCS options accepted by the assembler.

APCSC Specify additional APCS options for the compiler, other than those specified by INTERWORK and FPTYPE. Refer to Chapter 2 *The ARM Compilers* for details of the APCS options accepted by the compilers.

CC Specify the compiler to be used for the build. Select an appropriate value depending on the values of TARGET and INTERWORK:

`CC=$ARMCC`	For ARM libraries.
`CC=$TCC`	For Thumb or interworking options.

CCFLAGS Set CCFLAGS for additional compiler options. Refer to Chapter 2 *The ARM Compilers* for a full list of compiler options.

AS Specify the assembler to be used for the build:

Set AS to $(ARMASM) or $(TASM). Select an appropriate value depending on the values of TARGET and INTERWORK.

ASFLAGS Set ASFLAGS for additional assembler options. Refer to Chapter 5 *Assembler* for a full list of assembler options.

4.3 The ARM C++ libraries

This section describes ARM C++ library support. It provides information on how to use the standard C library from C++, and how to rebuild the C++ library. For detailed information on the contents of the C++ library, refer to:

- *Standard C++ library implementation definition* on page 3-35
- The Rogue Wave online documentation in `c:\ARM250\Html`.

4.3.1 Using the libraries

The ARM C++ library is provided with the same APCS, endian, and ARM/Thumb variants as the C library variants listed in *Precompiled ARM library variants* on page 4-6 and *Precompiled Thumb library variants* on page 4-5. For C++, the libraries are named `armcpplib_`*`apcs_variant`*.

You must build your code with APCS options that are supported by one of the precompiled library variants. The linker selects a precompiled library version that is compatible with the APCS options you select for the assembler and compiler. Refer to *Automatic inclusion of libraries* on page 6-39 for more information.

If you select APCS options for which a suitable precompiled library variant does not exist, the linker warns that the library file cannot be found.

Note

Because the ARM C++ library is supplied in binary form only, you cannot rebuild the library with non-supported APCS options.

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.

Using the standard C library from C++

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

C++ compiler helper functions

You can build a runtime version of the C++ helper functions from `c:\ARM250\Armcpplib_kit\cpplib\runtime.c`

From the DOS command-line, type:

```
armcpp -apcs APCS_options -DALL runtime.c
```

to build the `runtime.o` object file.

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

An ANSI C library binary

The ANSI C libraries are located in `c:\ARM250\lib`. Refer to *The ANSI C library* on page 4-5 for more information on the standard ANSI C libraries.

Using the standard C++ library

When you build your C++ code the linker selects an appropriate precompiled C++ library version.

The ARM C++ library provides:

Rogue Wave Standard C++ library

This is supplied in binary form only, as part of the precompiled armcpplib variants, and as sublibraries that you can use to rebuild armcpplib libraries.

The library files are located in `c:\ARM250\Armcpplib_kit\stdlib`.

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 `c:\ARM250\lib\Armcpplib_kit\cpplib`. The header files are installed in `c:\ARM250\include`.

The additional functions are:

- C++ compiler helper functions. Calls to these functions are generated by the compiler to implement certain language constructs.
- Support for the `new` operator. The header file `new.h` is installed in `c:\ARM250\include`. 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 `c:\ARM250\include`.

- Partial support for C++ iostreams. The header file `iostream.h` is installed in `c:\ARM250\include`. In addition, an identical header file named `stream.h` is installed in `c:\ARM250\include` for compatibility with previous releases of ARM C++.

4.3.2 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 `c:\ARM250\Armcpplib_kit\stdlib`, or the equivalent UNIX directory.
- Source code and rebuild scripts for the ARM C++ library additions. These are installed in `c:\ARM250\Armcpplib_kit\cpplib`, or the equivalent UNIX directory.

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

Follow these steps to rebuild a particular variant of the armcpplib library:

1. Change directory to:

   `Armcpplib_kit\cpplib\cpplib.b\`*`platform`*

   where *`platform`* is either `cchppa, gccsolrs, intelrel`, depending on your installation.

2. From the system command-line type:

   `armmake lib_cpplib_`*`variant`*

   where *`variant`* is the name of the library variant you want to build. If you are building under UNIX, type the name of your make utility instead of `armmake`.

   Variant names follow the suffix naming conventions described in *Library naming conventions* on page 4-4. The makefile supports the same APCS variants as the C library variants described in Table 4-1 on page 4-5 and Table 4-2 on page 4-6.

   For example, to build a little-endian ARM library with no software stack checking, and no frame pointer type:

   ```
   armmake lib_cpplib_cn.32l
   ```

   This creates `cpplib_cn.32l`.

   You can type `armmake all` to compile all 26 APCS variants.

3. Create a temporary directory and copy the new `cpplib` variant to it.

4. Copy the equivalent precompiled Rogue Wave library to your temporary directory. For the example in step 2, copy `Armcpplib_kit\stdlib\stdlib_cn.32l`.

5. Type the following commands to extract the object files from the two libraries and create a new armcpplib library:

```
armlib -e cpplib_cn.32l *
armlib -e stdlib_cn.32l *
armlib -c -o armcpplib_cn.32l *.o*
```

   If you are building under UNIX you must escape the * character. Type:

```
armlib -e cpplib_hc.32l \*
armlib -e stdlib_hc.32l \*
armlib -c -o armcpplib_hc.32l *.o*
```

   The new library is ready for use.

4.4 The embedded C library

The ARM embedded C library addresses problems caused by linking the standard ANSI C library into an embedded system. The ARM embedded C library is a subset of the full ANSI C library that addresses the following issues:

- The standard ANSI C library relies on underlying Angel SWIs for its operation. Unless your embedded system supports these SWIs in its SWI handler, the C library will not work correctly.
- For the standard ANSI C library to execute, the memory system must be configured in the way expected by the C library. This may not be easy to support in your embedded system.
- There is a minimum overhead of about 3KB when the standard ANSI C library is included. The embedded C library has no overhead.
- The ANSI C library is non-reentrant. This may cause problems in embedded systems.

Refer to *Example 3: Using the embedded C library* on page 10-21 of the *ARM Software Development Toolkit User Guide* for an example of how to use the embedded C library.

4.4.1 Embedded C library functions

The functions in the embedded C library are:

Runtime support functions

These functions carry out operations that are not available as ARM instructions, such as division. These functions are provided by the embedded C library.

Software floating-point library

When the compiler compiles code for use with software floating-point, it generates calls to routines in the library to perform floating-point operations. For example, to perform a double-precision multiply, the compiler generates a call to `_dmul`. All such routines are provided as standard by the embedded C library.

C library subset

This provides a subset of the C library routines. Only functions that fulfil the criteria described below have been included in the embedded C library. Refer to *C library subset* on page 4-20 for a complete list of the included functions.

Static data

The embedded C library does not make any use of static data. Because of this, it is automatically fully reentrant.

Operating-system independence

The embedded C functions do not rely on the underlying operating system in any way.

Many functions in the full ANSI library rely on the underlying OS to perform functions such as writing characters, or opening files. These functions are excluded from the embedded C library. For example, functions such as `printf()` are excluded, but `sprintf()` is not.

Standalone functions

Many functions in the full ANSI C library rely on a number of other functions in the C library to perform their operations. For example, `printf()` relies on functions such as `ferror()` and `fputc()`. This means that a single call to `printf()` includes a large amount of the C library code.

The embedded C library breaks many of these dependencies so that only the minimum amount of code needed to perform the operation is included.

C library subset

Table 4-4 lists the C library functions that are supported in the embedded C library.

Table 4-4 Supported C library functions

File	Functions				

`math.h`	`acos`
	`cos`
	`fmod`
	`modf`
	`tan`
		`asin`
		`cosh`
		`frexp`
		`pow`
		`tanh`
			`atan`
			`exp`
			`ldexp`
			`sin`
				`atan2`
				`fabs`
				`log`
				`sinh`
					`ceil`
					`floor`
					`log10`
					`sqrt`

`stdlib.h`	`abs`
	`calloc`
	`malloc`
	`strtoul`
		`atoi`
		`div`
		`qsort`
			`atol`
			`free`
			`realloc`
				`atof`
				`labs`
				`strtod`
					`bsearch`
					`ldiv`
					`strtol`

`ctype.h`	`isalnum`
	`islower`
	`isxdigit`
		`isalpha`
		`isprint`
		`tolower`
			`iscntrl`
			`ispunct`
			`toupper`
				`isdigit`
				`isspace`
					`isgraph`
					`isupper`

Table 4-4 Supported C library functions (Continued)

File	Functions				

`string.h`	`memchr`
	`strncpy`
	`strlen`
	`strspn`
		`memcmp`
		`strncmp`
		`strchr`
		`strstr`
			`memcpy`
			`strcat`
			`strcspn`
			`strxfrm`
				`memmove`
				`strcmp`
				`strncat`
				`strpbrk`
					`memset`
					`strcpy`
					`strrchr`

`stdio.h`	`sprintf`	`sscanf`			
`setjmp.h`	`setjmp`	`longjmp`			

4.4.2 Embedded C library variants

The following variants of the embedded C library are provided:

`embedded\armlib_i.16l`

Thumb little-endian interworking version with no software stack check and no frame pointer.

`embedded\armlib_i.16b`

Thumb big-endian interworking version with no software stack check and no frame pointer.

`embedded\armlib_cn.32l`

ARM little-endian non-interworking version with no software stack check and no frame pointer.

`embedded\armlib_cn.32b`

ARM big-endian non-interworking version with no software stack check and no frame pointer.

Building other variants

If you want to build different variants of the embedded C library (for example, to add software stack checking), see *Retargeting the ANSI C library* on page 4-6.

4.4.3 Callouts from the embedded C library

Because the embedded C library is designed to be completely independent of static data, or any operating system specific calls, it cannot perform operations that are operating system specific, or that reference static data directly. Instead, it performs a callout to a user-supplied function to perform these operations. There are four callouts that the embedded C library may make:

`__rt_trap()`

Called when an exception is generated in the embedded C library.

`__rt_errno_addr()`

Called to get the address of the variable `errno`.

`__rt_fp_status_addr()`

Called by the floating-point support code to get the address of the floating-point status word.

`__rt_heapdescriptor()`

Called by the heap storage management functions to get the heap descriptor.

In most cases, the embedded C library tests for the existence of the callout function before calling it. If the function does not exist, a default action is taken.

The callout functions and their default actions are described in the following sections.

4.4.4 __rt_trap

The embedded C library performs a callout to `__rt_trap()` to handle exceptions such as division by zero.

The following error codes may be generated by the embedded C library, and must be handled by `__rt_trap()`:

`0x80000020`	Integer divide by zero.
`0x80000200`	Invalid floating-point operation.
`0x80000201`	Floating-point overflow.
`0x80000202`	Floating-point divide by zero.

Syntax

```
void __rt_trap(ErrBlock *err, RegSet regs)
```

where:

`err` is a pointer to a block containing an error code followed by a zero-terminated string describing the error. `ErrBlock` is defined as:

```
typedef struct {
    unsigned ErrCode;
    char     ErrString[252];
} ErrBlock;
```

`regs` is a block of 16 words containing the values in the registers at the time of the exception. `RegSet` is defined as:

```
typedef unsigned RegSet[16];
```

If `__rt_trap()` is not defined, the embedded C library executes an undefined instruction.

4.4.5 __rt_errno_addr

This function is called to obtain the address of the C library `errno` variable when the embedded C library attempts to read or write `errno`. The embedded C library may set `errno` to:

`1, ERRDOM`	Input argument domain error.
`2, ERRANGE`	Result range error.

Syntax

```
volatile int *__rt_errno_addr(void)
```

If `__rt_errno_addr()` is not defined, the embedded C library does not attempt to set or read `errno`.

4.4.6 __rt_fp_status_addr

This function returns the address of the floating-point status register. Example 4-2 shows how__rt_fp_status_addr() is defined.

Example 4-2

```
/*
Descriptions of these bits may be found on page 9-8 of the 7500FE data sheet. The
software floating-point library does not implement the IXE or UFE exceptions
*/
#define IOC_Bit     (1 << 0)    /* Invalid op cumulative */
#define DZC_Bit     (1 << 1)    /* Divide zero cumulative */
#define OFC_Bit     (1 << 2)    /* Overflow cumulative */
#define UFC_Bit     (1 << 3)    /* Underflow cumulative */
#define IXC_Bit     (1 << 4)    /* Inexact cumulative */
#define ND_Bit      (1 << 8)    /* No denormalised numbers */
#define IOE_Bit     (1 << 16)   /* Invalid operation exception */
#define DZE_Bit     (1 << 17)   /* Divide zero exception */
#define OFE_Bit     (1 << 18)   /* Overflow exception */
#define UFE_Bit     (1 << 19)   /* Underflow exception */
#define IXE_Bit     (1 << 20)   /* Inexact exception */
#define FP_SW_LIB   0x40000000
#define FP_SW_EMU   0x01000000
#define FP_HW_FPA   0x81000000

/* This enables all supported exceptions. It is useful to have them enabled by
default. */

static unsigned __fp_status_flags = FP_SW_LIB+IOE_Bit+DZE_Bit+OFE_Bit;
unsigned *__rt_fp_status_addr(void)
{
    return &__fp_status_flags;
}
```

If __rt_fp_status_addr() is not defined, the embedded C library does not attempt to read or write the floating-point status register. For the purposes of raising exceptions, it assumes that the following exception enable bits are set:

```
const unsigned __fp_status_default =
                    FP_SW_LIB+IOE_Bit+DZE_Bit+OFE_Bit;
```

4.4.7 __rt_embeddedalloc_init

The heap manager manages a single contiguous area of memory. To initialize the heap manager, declare and call the following function from the startup code of your embedded system.

Syntax

```
void *__rt_embeddedalloc_init(void *base, size_t size)
```

Return

This function returns a pointer to a heap descriptor. You must call this function before any other heap functions are used.

4.4.8 __rt_heapdescriptor

This function is called from the storage management functions. The value returned by this function must be the same as that returned by the call to `__rt_embeddedalloc_init()`. This function is not optional, because the heap cannot work without the heap descriptor.

Syntax

```
void *__rt_heapdescriptor(void)
```

4.5 Target-dependent ANSI C library functions

Implementation of the following ANSI standard functions fully depends on the target operating system. None of these functions are used internally by the library. This means that, if any of these functions are not implemented, only clients that directly call the function will fail.

The target-dependent ANSI C library functions are:

- `clock()`
- `_clock_init()`
- `getenv()`
- `_getenv_init()`
- `remove()`
- `rename()`
- `system()`
- `time()`.

4.5.1 clock

The standard C library clock function from `time.h`.

Syntax

```
clock_t clock(void)
```

Implementation

The compiler is expected to predefine `__CLK_TCK` if the units of `clock_t` differ from the default of centiseconds. If this is not done, you must edit `time.h` to define appropriate values for `CLK_TCK` and `CLOCKS_PER_SEC`.

4.5.2 _clock_init

An optional initialization function for `clock()`.

Syntax

```
___weak void _clock_init(void)
```

Implementation

This function is declared weak. You should initialize `clock()` if it must work with a read-only timer. If implemented, `_clock_init()` is called from the library initialization code.

4.5.3 getenv

The standard C library `getenv()` function from `stdlib.h`.

Syntax

```
char *getenv(const char *string)
```

4.5.4 _getenv_init

An optional initialization function for `getenv()`.

Syntax

```
___weak void _getenv_init(void)
```

Implementation

This function is declared weak. If it is implemented, it is called from the library initialization code.

4.5.5 remove

The standard C library `remove()` function from `stdio.h`.

Syntax

```
int remove(const char *pathname)
```

4.5.6 rename

The standard C library `rename()` function from `stdio.h`.

Syntax

```
int rename(const char *old, const char *new)
```

4.5.7 system

The standard C library `system()` function from `stdlib.h`.

Syntax

```
int system(const char *string)
```

4.5.8 time

The standard C library `time()` function from `time.h`.

Syntax

```
time_t time(time_t *timer)
```

4.6 Target-dependent I/O support functions

The C library, as supplied, only conveniently handles byte-stream files. This means that handling other file types in the target-independent I/O support code can be done, but may be complicated to implement. For example, block stream files are simple to support in the absence of user-supplied buffers.

If any I/O function is used, `hostsys.h` must define the type `FILEHANDLE`. The value of `FILEHANDLE` is returned by `_sys_open()` and identifies an open file on the host system. There must be at least one distinguished value of type `FILEHANDLE`. It is defined by the macro `NONHANDLE`, and is used to distinguish a failed call to `_sys_open()`.

For an unaltered `__rt_lib_init()`, the macro `TTYFILENAME` must be defined as a string to be used in opening a file to terminal.

The macro `HOSTOS_NEEDSENSURE` should be defined if the host operating system requires an ensure operation to flush OS file buffers to disk if an OS write is followed by an OS read that itself requires a seek (the flush happens before the seek).

The target-dependent input and output functions are:

- `_sys_open()`
- `_sys_close()`
- `_sys_read()`
- `_sys_write()`
- `_sys_ensure()`
- `_sys_flen()`
- `_sys_iserror()`
- `_sys_istty()`
- `_sys_tmpnam()`
- `_sys_ttywrch`.

4.6.1 _sys_open

Open a file.

Syntax

```
FILEHANDLE _sys_open(const char *name, int openmode)
```

Implementation

This function is required by `fopen()` and `freopen()`. These in turn are required if any I/O function is to be used.

The *`openmode`* parameter is a bitmap, in which the bits mostly correspond directly to the ANSI mode specification. Refer to `hostsys.h` for details. Target-dependent extensions are possible, in which case `freopen()` must also be extended.

4.6.2 _sys_close

Close a file previously opened with `_sys_open()`.

Syntax

```
int _sys_close(FILEHANDLE fh)
```

Implementation

This function must be defined if any I/O function is to be used. The return value is 0 or an error indication.

4.6.3 _sys_read

Read the contents of a file into a buffer.

Syntax

```
int _sys_read (FILEHANDLE fh, unsigned  char *buf, unsigned len,
               int mode)
```

Implementation

This function must be defined if any input function or `scanf()` variant is to be used. The *`mode`* argument is a bitmap describing the state of the FILE connected to *`fh`*, as for `_sys_write()`. The return value is one of the following:

- the number of characters *not* read (that is, *`len`* - *`result`* were read)
- an error indication
- an EOF indicator. The EOF indication involves the setting of 0x80000000 in the normal result. The target-independent code is capable of handling either:

  early EOF where the last read from a file returns some characters plus an EOF indicator

  late EOF where the last read returns just EOF:

  `int _sys_seek(FILEHANDLE fh, long pos)`

  The function must be defined if any input or output function is to be used. It puts the file pointer at offset *`pos`* from the beginning of the file. The result is >= 0 if okay, and is negative for an error.

4.6.4 _sys_write

Write the contents of a buffer to a file previously opened with `_sys_open()`.

Syntax

```
int _sys_write(FILEHANDLE fh, const unsigned char *buf,
               unsigned len, int mode)
```

Implementation

This function must be defined if any output function or `printf()` variant is to be used. The *`mode`* parameter is a bitmap describing the state of the FILE connected to *`fh`*. See the `_IOxxx` constants in `ioguts.h` for its meaning. Only a few of these bits are expected to be needed by `_sys_write()`.

The return value is the number of characters *not* written (that is, non-0 denotes a failure of some sort), or a negative error indicator.

4.6.5 _sys_ensure

Flush buffers associated with a file.

Syntax

```
int _sys_ensure(FILEHANDLE fh)
```

Implementation

This function is required only if you define `HOSTOS_NEEDSENSURE`. A call to `_sys_ensure()` flushes any buffers associated with *`fh`*, and ensures that the file is up to date on the backing store medium. The result is >= 0 if okay, and is negative for an error.

4.6.6 _sys_flen

Return the current length of a file.

Syntax

```
long _sys_flen(FILEHANDLE fh)
```

Implementation

This function returns the current length of the file *fh*, or a negative error indicator. It is needed in order to convert `fseek(, SEEK_END)` into `(, SEEK_SET)` as required by `_sys_seek()`. It must be defined if `fseek()` is to be used. You can adopt a different model here if the underlying system directly supports seeking relative to the end of a file, in which case `_sys_flen()` can be eliminated.

4.6.7 _sys_iserror

This function determines if the return value for any of the _sys functions is an error.

Syntax

```
int _sys_iserror(int status)
```

Implementation

A `_sys_iserror()` function, or a `_sys_iserror()` macro, is required if any of the `_sys` functions that return an **`int`** value are implemented.

4.6.8 _sys_istty

Determine if a file is connected to a terminal.

Syntax

```
int _sys_istty(FILE *f)
```

Implementation

This function must return non-zero if the argument file is connected to a terminal.

This function is used to provide default unbuffered behavior (in the absence of a call to `set(v)buf`), and to disallow seeking. It must be defined if any output function, including `sprintf()` variants, or `fseek()` is to be used.

4.6.9 _sys_tmpnam

This function converts a file number (fileno) for a temporary file into a unique filename, such as `tmp0001`.

Syntax

```
void _sys_tmpnam(char *name, int fileno)
```

Implementation

This function must be defined if `tmpnam()` or `tmpfil()` are to be used. It returns the filename in *`name`*.

4.6.10 _ttywrch

Write a character, notionally to the console.

Syntax

```
void _ttywrch(int ch)
```

Implementation

This function is required. It is used in the host-independent part of the library in the last-ditch error reporter, when writing to stderr is believed to have failed or to be unsafe. For example, it is used in the default SIGSTK handler.

4.7 Target-dependent kernel functions

The Kernel handles the entry to, and exit from, an application linked with the library. It also exports some variables for use by other parts of the library. Details of what the kernel must do depend on the target environment.

You can use the ARMulator-host C library kernel in `cl\angel\kernel.s` as a prototype.

The target-dependent kernel functions are:

- `__main()`
- `__rt_exit()`
- `__rt_command_string()`
- `__rt_trap()`
- `__rt_alloc()`
- `__rt_malloc()`
- `__rt_free()`.

4.7.1 __main

This function provides the entry point to the application.

Syntax

```
__main()
```

Implementation

This function is called after low-level library initialization. The initialization required depends on the target environment. It may include:

- Initializing heap, stack, and fp support.
- Calling appropriate `__osdep_xxx_init()` functions if they have been implemented. Refer to *Target-dependent operating system functions* on page 4-41 for more information.

`__main()` must call `__rt_lib_init()` to initialize the body of the library. Refer to *Initializing the runtime libraries* on page 4-10 for more information.

4.7.2 __rt_exit

Finalize library initialization.

Syntax

```
void __rt_exit(int ret_code)
```

Implementation

This mandatory function finalizes the library, not including calling `atexit()` handlers. It returns to the operating system with its argument as a completion code. It is called by `__main()` or `__exit()`.

4.7.3 __rt_command_string

Return the command-line string used to invoke the program.

Syntax

```
char *__rt_command_string(char *str, int len)
```

Implementation

This mandatory function returns the address of the string used to invoke the program. The command-line is stored in buffer *`str`* of length *`len`* if the buffer is large enough to store it.

4.7.4 __rt_trap

Handle a fault, such as the processor detected a trap and enabled fp exception.

Syntax

```
void __rt_trap(ErrBlock *err, RegSet regs)
```

where:

err is a pointer to a block containing an error code followed by a zero-terminated string describing the error. `ErrBlock` is defined as:

```
typedef struct {
    unsigned ErrCode;
    char     ErrString[252];
} ErrBlock;
```

regs is a block of 16 words containing the registers at the time of the exception. `RegSet` is defined as:

```
typedef unsigned RegSet[16];
```

If `__rt_trap()` is not defined, the embedded C library executes an undefined instruction.

Implementation

This function is mandatory. The argument register set describes the processor state at the time of the fault, with the pc value addressing the faulting instruction (except perhaps in the case of imprecise floating-point exceptions). The implementation in the ARMulator kernel is usually adequate.

4.7.5 __rt_alloc

The low-level memory allocator underlying `malloc()`.

Syntax

The syntax is:

```
unsigned __rt_alloc(unsigned minwords, void **block)
```

Implementation

The `malloc()` function allocates memory only between HeapBase and HeapTop. A call to `__rt_alloc()` attempts to move HeapTop.

`__rt_alloc()` should try to allocate a block of a size greater than or equal to *`minwords`*. If this is not available, and if `__osdep_heapsupport_extend()` is defined, it should be called to attempt to move HeapTop. Otherwise (or if the call fails) it should allocate the largest possible block of sensible size.

The return value is the size of block allocated, and `*`*`block`* is set to point to the start of the allocated block. The return value may be 0 if no sensibly-sized block can be allocated. Allocations are rounded up to a suitable size to avoid an excessive number of calls to `__rt_alloc()`.

4.7.6 __rt_malloc

A function pointer to a primitive memory allocation function.

Syntax

```
void *(*__rt_malloc)(size_t n)
```

Implementation

The kernel should initialize this function pointer to point to a primitive memory allocation function. The library itself contains no calls to `malloc()` other than those from functions of the malloc family, such as `calloc()`. The function pointed to by `__rt_malloc()` is called instead.

If `malloc()` is linked into the image, `__rt_malloc()` is set to `malloc()` during initialization. Otherwise it is set to `__malloc()`. The use of `__rt_malloc()` ensures that allocations succeed if they are made before `malloc()` is initialized, and prevents `malloc()` from being linked into an image if it is not used.

4.7.7 __rt_free

A function pointer to a primitive memory freeing function.

Syntax

```
extern void (*__rt_free)(void *)
```

Implementation

The kernel should initialize this function pointer to point to a primitive memory freeing function. See *__rt_malloc* on page 4-39.

4.8 Target-dependent operating system functions

The target-dependent operating system functions are:

- `__osdep_traphandlers_init()`
- `__osdep_traphandlers_finalise()`
- `__osdep_heapsupport_init()`
- `__osdep_heapsupport_finalise()`
- `__osdep_heapsupport_extend()`
- `_hostos_error_string()`
- `_hostos_signal_string()`.

4.8.1 __osdep_traphandlers_init

Install a handler to catch processor aborts and pass them to `__rt_trap()` .

Syntax

```
void __osdep_traphandlers_init(void)
```

4.8.2 __osdep_traphandlers_finalise

Remove the processor handlers installed by the `__osdep_traphandlers_init()` function.

Syntax

```
void __osdep_traphandlers_finalise(void)
```

4.8.3 __osdep_heapsupport_init

Initialize heap support.

Syntax

```
void __osdep_heapsupport_init(HeapDescriptor *hd)
```

Implementation

This function must be provided, but may be empty. It is called when the heap is being initialized.

4.8.4 __osdep_heapsupport_finalise

Finalize heap support.

Syntax

```
void __osdep_heapsupport_finalise(void)
```

Implementation

This function must be provided, but may be empty. It is called when the heap is being finalized.

4.8.5 __osdep_heapsupport_extend

Extend the heap.

Syntax

```
__value_in_regs struct ExtendResult
__osdep_heapsupport_extend(int size, HeapDescriptor *hd)
```

where *ExtendResult* is a structure of the form:

```
struct ExtendResult{
    int acqsize;
    void * acqbase;
    };
```

Implementation

This function requests extension of the heap by at least *size* bytes. The return values are the number of bytes acquired, and the base address of the new acquisition. This function must be provided, but an empty version that returns:

```
struct{
       0;
       NULL;}
```

is sufficient if heap extension is not needed.

4.8.6 _hostos_error_string

Return an error message.

Syntax

```
char *_hostos_error_string(int no, char *buf)
```

Implementation

This function is called to return a string describing an error outside the set `ERRxxx` defined in `errno.h`. It may generate the message into the supplied *`buf`* if it needs to do so. It must be defined if `perror()` or `strerror()` is to be used.

4.8.7 _hostos_signal_string

Return a signal description.

Syntax

```
char *_hostos_signal_string(int no)
```

Implementation

This function is called to return a string describing a signal whose number is outside the set `SIGxxx` defined in `signal.h`.

Chapter 5
Assembler

This chapter describes the language features that are provided by the ARM assembler, such as pseudo-instructions, directives and macros. It contains the following sections:

See Table 5-1 on page 5-2 to locate individual pseudo-instructions or directives.

This chapter does not contain detailed information on how to write ARM assembly language. Refer to Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for tutorial information on how to use many of the language features described here.

For detailed information on ARM and Thumb instruction mnemonics, refer to the *ARM Architectural Reference Manual* and the *ARM FPA10 Data Sheet*.

Table 5-1 Directives and pseudo-instructions

Directives:	**Reporting:**	**Assembly control:**
AREA on page 5-38	ASSERT on page 5-40	[or IF on page 5-33
CODE16 on page 5-42	INFO or ! on page 5-67	| or ELSE on page 5-34
CODE32 on page 5-43	OPT on page 5-77	] or ENDIF on page 5-34
END on page 5-55	SUBT on page 5-85	GET or INCLUDE on page 5-63
ENTRY on page 5-56	TTL on page 5-86	INCBIN on page 5-66
NOFP on page 5-76	**Symbol definition:**	MACRO on page 5-73
ROUT on page 5-81	CN on page 5-41	MEND on page 5-75
Data definition:	CP on page 5-44	MEXIT on page 5-75
# on page 5-31	EQU or * on page 5-57	WEND on page 5-86
% on page 5-32	EXPORT or GLOBAL on page 5-58	WHILE on page 5-87
^ or MAP on page 5-35	FN on page 5-59	
ALIGN on page 5-36	GBLA on page 5-60	**ARM pseudo-instructions:**
DATA on page 5-45	GBLL on page 5-61	ADR on page 5-12
DCB or = on page 5-46	GBLS on page 5-62	ADRL on page 5-13
DCD or & on page 5-47	IMPORT or EXTERN on page 5-64	LDFD on page 5-14
DCDU on page 5-48	KEEP on page 5-68	LDFS on page 5-15
DCFD on page 5-49	LCLA on page 5-69	LDR on page 5-16
DCFDU on page 5-50	LCLL on page 5-70	NOP on page 5-18
DCFS on page 5-51	LCLS on page 5-71	
DCFSU on page 5-52	RLIST on page 5-79	**Thumb pseudo-instructions:**
DCW on page 5-53	RN on page 5-80	ADR on page 5-20
DCWU on page 5-54	SETA on page 5-82	LDR on page 5-21
LTORG on page 5-72	SETL on page 5-83	MOV on page 5-23
	SETS on page 5-84	NOP on page 5-24

5.1 Command syntax

Note

The ARM assembler, armasm, assembles both ARM and Thumb assembly languages. The obsolete Thumb assembler, tasm, is provided in the Software Development Toolkit for backwards compatibility only.

Invoke the ARM assembler using this command:

```
armasm [-apcs [none | 3[/qualifier[/qualifier[...]]]]]
[-arch architecture] [-bigend | -littleend] [-checkreglist]
[-cpu ARMcore] [-depend dependfile] [-errors errorfile] [-g]
[-help] [-keep] [-i dir [,dir]…] [-list listingfile [options]]
[-maxcache n] [-MD-] [-nocache] [-noesc] [-noregs] [-nowarn]
[-o filename] [-predefine "directive"] [-processor ARMcore]
[-unsafe] [-via file] [-16 | -32] inputfile
```

where:

`-apcs [none | 3[/qualifier[/qualifier]]]`

specifies whether you are using the ARM Procedure Call Standard or not, and may specify some attributes of code areas. See Chapter 6 *Using the Procedure Call Standards* in the *ARM Software Development Toolkit User Guide* for more information.

`none` specifies that *`inputfile`* does not use APCS. APCS registers are not set up. Qualifiers are not allowed.

`3` specifies that *`inputfile`* uses APCS version 3. APCS registers are set up. This is the default.

Values for *`qualifier`* are:

`nointerwork`

specifies that the code is not suitable for ARM/Thumb interworking. This is the default.

`interwork`

specifies that the code is suitable for ARM/Thumb interworking. This option has the same effect as specifying the `INTERWORK` attribute for all code areas in the source files to be assembled. Refer to the *ARM Software Development Toolkit User Guide* for more information on ARM/Thumb interworking.

`swstackcheck`
: specifies that the code in *`inputfile`* carries out software stack checking.

`noswstackcheck`
: specifies that the code in *`inputfile`* does not carry out software stack-limit checking. This is the default.

`reentrant`
: specifies that the code in *`inputfile`* is reentrant. This option is obsolete and is provided for backwards compatibility only. The SDT version 2.50 linker does not link objects assembled with `-apcs /reentrant`.

`nonreentrant`
: specifies that the code in *`inputfile`* is non reentrant. This is the default.

`fp`
: specifies that the code in *`inputfile`* uses a frame pointer. This option is obsolete and is provided for backwards compatibility only.

`nofp`
: specifies that the code in *`inputfile`* does not use a frame pointer. This is the default.

`-arch` *`architecture`*
: sets the target architecture. Some processor-specific instructions produce either errors or warnings if assembled for the wrong target architecture. See also the `-unsafe` assembler option. Valid values for *`architecture`* are `3`, `3m`, `4`, `4T`, `4TxM`.

`-bigend`
: instructs the assembler to assemble code suitable for a big-endian ARM. This option sets the built-in variable `{ENDIAN}` to `big`. The default is `-littleend`.

`-checkreglist`
: instructs the assembler to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order. If this is not the case, a warning is given.

`-cpu` *`ARMcore`*
: sets the target ARM core. Valid values for *`ARMcore`* are:

`ARM6`	an ARM 6 family processor.
`ARM7`	an ARM 7 family processor.

`ARM7M` an ARM 7 family processor with fast multiplier.

`ARM7T` an ARM 7 family processor with Thumb.

`ARM7TDI` an ARM 7 family processor with Thumb and debug extensions.

`ARM7TDMI` an ARM 7 family processor with Thumb, debug and fast multiplier.

`ARM7TM` an ARM 7 family processor with Thumb and fast multiplier.

`ARM8` an ARM 8 family processor.

`ARM9` an ARM 9 family processor.

`ARM9TM` an ARM 9 family processor with Thumb and fast multiplier.

`StrongARM1` a StrongARM1 processor.

`SA-110` an SA-110 processor.

`-depend` *`dependfile`*

instructs the assembler to save source file dependency lists. These are suitable for use with make utilities.

`-errors` *`errorfile`*

instructs the assembler to output error messages to *`errorfile`*.

`-g` instructs the assembler to generate debug tables. Use the following command-line options to control the behavior of `-g`:

`-dwarf` to select DWARF1 debug tables. This option is obsolete. Use `-dwarf2` or `-dwarf1`.

`-dwarf1` to select DWARF1 debug tables. This option is not recommended for C++.

`-dwarf2` to select DWARF2 debug tables. This is the default and is selected if `-g` with no dwarf option is specified.

`-help` instructs the assembler to display a summary of the assembler command-line options.

`-keep` instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger. See *KEEP directive* on page 5-68.

`-i` *`dir`* `[,`*`dir`*`]…`

adds directories to the source file search path so that arguments to `GET`/`INCLUDE` directives do not need to be fully qualified. See *GET or INCLUDE directive* on page 5-63 and *INCBIN directive* on page 5-66.

`-list` *`listingfile options`*

instructs the assembler to output a detailed listing of the assembly language produced by the assembler to *`listingfile`*. Use the following command-line options to control the behavior of `-list`:

`-noterse`

turns the `terse` flag off. When this option is on, lines skipped due to conditional assembly do not appear in the listing. If the `terse` option is off, these lines do appear in the listing. The default is on.

`-width` sets the listing page width. The default is 79 characters.

`-length` sets the listing page length. Length zero means an unpaged listing. The default is 66 lines.

`-xref` instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros. The default is off.

`-littleend`

instructs the assembler to assemble code suitable for a little-endian ARM. This option sets the built-in variable `{ENDIAN}` to `little`. This is the default.

`-maxcache` *`n`*

sets the maximum source cache size to *n*. The default is 8MB.

`-MD-` is for the use of the ARM Project Manager. It instructs the assembler to write makefile dependencies to the Project Manager.

`-nocache` turns off source caching. By default the assembler caches source files on the first pass and reads them from memory on the second pass.

`-noesc` instructs the assembler to ignore C-style escaped special characters, such as `\n` and `\t`.

`-noregs` instructs the assembler not to predefine register names. Refer to *Predefined register and coprocessor names* on page 5-9 for a list of predefined register names.

`-nowarn` turns off warning messages.

`-o` *`filename`*

names the output object file. If this option is not specified, the assembler uses the second command-line argument that is not a valid command-line option as the name of the output file. If there is no such argument, the assembler creates an object filename of the form *`inputfilename`*`.o`

`-predefine "`*`directive`*`"`

instructs the assembler to pre-execute one of the `SET` directives. You must enclose *`directive`* in double quotes. See:

- *SETA directive* on page 5-82.
- *SETL directive* on page 5-83.
- *SETS directive* on page 5-84.

The assembler executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value. Arguments to `SETS` must be enclosed in escaped double quotation marks, for example:

```
-pd "Version SETS \"beta-4\""
-pd "VersionNum SETA 4"
```

`-processor` *`ARMcore`*

is a synonym for `-cpu`

`-unsafe` allows assembly of a file containing instructions that are not available on the specified architecture and processor. Corresponding error messages are changed to warning messages.

`-via` *`file`* instructs the assembler to open *`file`* and read in command-line arguments to the assembler.

`-16` instructs the assembler to interpret instructions as Thumb instructions. This is equivalent to placing a `CODE16` directive at the head of the source file.

`-32` instructs the assembler to interpret instructions as ARM instructions. This is the default.

`inputfile` specifies the input file for the assembler. Input files must be ARM or Thumb assembly language source files.

5.2 Format of source lines

The general form of source lines in an ARM assembly language module is:

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`

All three sections of the source line are optional. Instructions cannot start in the first column. They must be preceded by white space even if there is no preceding symbol.

You can use blank lines to make your code more readable.

`symbol` is usually a label. See *Labels* on page 5-27. In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. This is made clear in the description of the directive.

`symbol` must begin in the first column and cannot contain any whitespace character such as a space or a tab. See *Symbol naming rules* on page 5-25.

5.3 Predefined register and coprocessor names

The following register names and coprocessor names are predefined by the ARM assembler. All register and coprocessor names are case-sensitive.

5.3.1 Predeclared register names

The following register names are predeclared:

- `R0-R15`
- `r0-r15`
- `a1-a4`
- `v1-v8`
- `sp` and `SP`
- `lr` and `LR`
- `pc` and `PC`
- `sl` and `SL`

Note

`fp`, `FP`, `ip`, `IP`, `sb`, and `SB` are also predeclared. This is for backwards compatibility only.

5.3.2 Predeclared program status register names

The following program status register names are predeclared:

- `cpsr` and `CPSR`
- `spsr` and `SPSR`

5.3.3 Predeclared floating-point register names

The following floating-point register names are predeclared:

- `f0-f7`
- `F0-F7`

5.3.4 Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

- `p0-p15`
- `c0-c15`

5.4 Built-in variables

Table 5-2 lists the built-in variables defined by the ARM assembler.

Built-in variables cannot be set using the `SETA`, `SETL`, or `SETS` directives. They can be used in expressions or conditions, for example:

```
        IF {ARCHITECTURE} = "4T"
```

Table 5-2 Built-in variables

`{PC}` or `.`	Address of current instruction.
`{VAR}` or `@`	Current value of the storage area location counter.
`{TRUE}`	Logical constant true.
`{FALSE}`	Logical constant false.
`{OPT}`	Value of the currently-set listing option. The `OPT` directive can be used to save the current listing option, force a change in it, or restore its original value.
`{CONFIG}`	Has the value `32` if the assembler is in ARM mode, and the value `16` if it is in Thumb mode.
`{ENDIAN}`	Has the value `big` if the assembler is in big-endian mode, and the value `little` if it is in little-endian mode.
`{CODESIZE}`	Has the value `16` if compiling Thumb code. Otherwise, `32`.
`{CPU}`	Has the name of the selected cpu, or `generic ARM` if no cpu has been specified.
`{ARCHITECTURE}`	Has the value of the selected ARM architecture: • `3` • `3M` • `4` • `4T` • `4TxM`

`{PCSTOREOFFSET}`	Is the offset between the address of the
	`STR pc,[...]`
	or
	`STM Rb,{... pc}`
	instruction and the value of pc stored out. This varies depending on the CPU and architecture specified.

5.5 ARM pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

The pseudo-instructions available in ARM state are described in the following sections:

- *ADR ARM pseudo-instruction* on page 5-12
- *ADRL ARM pseudo-instruction* on page 5-13
- *LDFD ARM pseudo-instruction* on page 5-14
- *LDFS ARM pseudo-instruction* on page 5-15
- *LDR ARM pseudo-instruction* on page 5-16
- *NOP ARM pseudo-instruction* on page 5-18.

Refer to *Thumb pseudo-instructions* on page 5-19 for information on pseudo-instructions that are available in Thumb state.

5.5.1 ADR ARM pseudo-instruction

The `ADR` pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

The syntax of `ADR` is:

`ADR{`*`condition`*`}` *`register`*`,`*`expression`*

where:

`register` is the register to load.

`expression`

is a program-relative or register-relative expression that evaluates to:

- a non word-aligned address within 255 bytes
- a word-aligned address within 1020 bytes.

The address can be either before or after the address of the instruction or the base register.

See *Register-relative and program-relative expressions* on page 5-89.

Usage

`ADR` always assembles to one instruction. The assembler attempts to produce a single `ADD` or `SUB` instruction to load the address. If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

If *`expression`* is program-relative, it must evaluate to an address in the same code area as the `ADR` pseudo-instruction. Otherwise the address may be out of range after linking.

Example

```
start   MOV     r0,#10
        ADR     r4,start        ; => SUB r4,pc,#0xc
```

5.5.2 ADRL ARM pseudo-instruction

The `ADRL` pseudo-instruction loads a program-relative or register-relative address into a register. It is similar to the `ADR` pseudo-instruction. `ADRL` can load a wider range of addresses than `ADR` because it generates two data processing instructions.

Syntax

The syntax of `ADRL` is:

`ADRL{`*`condition`*`}` *`register`*`,`*`expression`*

where:

`register` is the register to load.

`expression` is a register-relative or program-relative expression that evaluates to:

- a non word-aligned address within 64KB
- a word-aligned address within 256KB.

The address can be either before or after the address of the instruction or the base register. See *Register-relative and program-relative expressions* on page 5-89.

Usage

`ADRL` always assembles to two instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. See *LDR ARM pseudo-instruction* on page 5-16 for information on loading a wider range of addresses. See also Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide*.

If *`expression`* is program-relative, it must evaluate to an address in the same code area as the `ADRL` pseudo-instruction. Otherwise the address may be out of range after linking.

Note

`ADRL` is not available when assembling Thumb instructions. Use it only in ARM code.

Example

```
start   MOV     r0,#10
        ADRL    r4,start + 60000    ; => ADD r4,pc,#0xe800
                                    ;  ADD r4,r4,#0x254
```

5.5.3 LDFD ARM pseudo-instruction

The `LDFD` pseudo-instruction loads a floating-point register with a double precision floating-point constant.

Note

You can use `LDFD` only if your system has a Floating Point Accelerator, or software that emulates one.

This section describes the `LDFD` *pseudo*-instruction only. Refer to the *ARM FPA10 Data Sheet* for information on the `LDFD` *instruction*.

Syntax

The syntax of `LDFD` is:

```
LDFD{condition} fp-register,=expression
```

where:

condition is an optional condition code.

fp-register

is the floating-point register to be loaded.

expression

evaluates to a floating-point constant. The assembler places the constant in a literal pool and generates a program-relative `LDFD` instruction to read the constant from the literal pool. Two words are used to store the constant in the literal pool.

The offset from pc to the constant must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

Usage

The range for double precision numbers is:

- Maximum 1.79769313486231571e+308
- Minimum 2.22507385850720138e–308.

Example

```
        LDFD    f1,=3.12E106    ; loads 3.12E106 into f1
```

5.5.4 LDFS ARM pseudo-instruction

The `LDFS` pseudo-instruction loads a floating-point register with a single precision floating-point constant.

Note

You can use `LDFS` only if your system has a Floating Point Accelerator, or software that emulates one.

This section describes the `LDFS` *pseudo*-instruction only. Refer to the *ARM FPA10 Data Sheet* for information on the `LDFS` *instruction*.

Syntax

The syntax of `LDFS` is:

`LDFS{`*`condition`*`}` *`fp-register`*`,=`*`expression`*

where:

`condition` is an optional condition code.

`fp-register`

is the floating-point register to be loaded.

`expression`

evaluates to a floating-point constant. The assembler places the constant in a literal pool and generates a program-relative `LDFS` instruction that reads the constant from the literal pool.

The offset from the pc to the constant must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

Usage

The range for single precision values is:

- Maximum 3.40282347e+38F
- Minimum 1.17549435e–38F.

Example

```
        LDFS    f1,=3.12E-6     ; loads 3.12E-6 into f1
```

5.5.5 LDR ARM pseudo-instruction

The `LDR` pseudo-instruction loads a register with either:

- a 32-bit constant value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only. Refer to the *ARM Architectural Reference Manual* for information on the `LDR` *instruction*.

Syntax

The syntax of `LDR` is:

```
LDR{condition} register,=[expression | label-expression]
```

where:

`condition` is an optional condition code.

`register` is the register to be loaded.

`expression`

evaluates to a numeric constant:

- If the value of *`expression`* is within range of a `MOV` or `MVN` instruction, the assembler generates the appropriate instruction.
- If the value of *`expression`* is *not* within range of a `MOV` or `MVN` instruction, the assembler places the constant in a literal pool and generates a program-relative `LDR` instruction that reads the constant from the literal pool.

  The offset from the pc to the constant must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

`label-expression`

is a program-relative or external expression. The assembler places the value of *`label-expression`* in a literal pool and generates a program-relative `LDR` instruction that loads the value from the literal pool.

The offset from the pc to the value in the literal pool must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

If *label-expression* is an external expression, or is not contained in the current area, the assembler places a linker relocation directive in the object file. The linker ensures that the correct address is generated at link time.

Usage

The LDR pseudo-instruction is used for two main purposes:

- to generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV and MVN instructions.
- to load a program-relative or external address into a register. The address remains valid regardless of where the linker places the AOF area containing the LDR.

Refer to Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

Example

```
LDR     r1,=0xfff       ; loads 0xfff into r1
                        ;
LDR     r2,=place       ; loads the address of
                        ; place into r2
```

5.5.6 NOP ARM pseudo-instruction

`NOP` generates the preferred ARM no-operation code. This is:

```
MOV r0,r0
```

Syntax

The syntax of `NOP` is:

```
NOP
```

Usage

`NOP` cannot be used conditionally. Not executing a no-operation is the same as executing it, so conditional execution is not required.

Condition codes are unaltered by `NOP`.

5.6 Thumb pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

The pseudo-instructions that are available in Thumb state are in the following sections:

- *ADR Thumb pseudo-instruction* on page 5-20
- *LDR Thumb pseudo-instruction* on page 5-21
- *MOV Thumb pseudo-instruction* on page 5-23
- *NOP Thumb pseudo-instruction* on page 5-24.

Refer to *ARM pseudo-instructions* on page 5-11 for information on pseudo-instructions that are available in ARM state.

5.6.1 ADR Thumb pseudo-instruction

The `ADR` pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

The syntax of `ADR` is:

`ADR` *`register`*`,` *`expression`*

where:

`register` is the register to load.

`expression`

is a register-relative or program-relative expression that evaluates to a word-aligned address within the range +4 to +1020 bytes. *`expression`* must be defined locally, it cannot be imported.

Refer to *^ or MAP directive* on page 5-35 for more information on register-relative expressions.

Usage

In Thumb state, `ADR` can generate word-aligned addresses only. Use the `ALIGN` directive to ensure that *`expression`* is aligned.

If *`expression`* is program-relative, it must evaluate to an address in the same code area as the `ADR` pseudo-instruction. There is no guarantee that the address will be within range after linking if it resides in another AOF area.

Example

```
        ADR     r4,txampl       ; => ADD r4,pc,#nn
        ; code
        ALIGN
txampl  DCW     0,0,0,0
```

5.6.2 LDR Thumb pseudo-instruction

The `LDR` pseudo-instruction loads a low register with either:

- a 32-bit constant value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only. Refer to the *ARM Architectural Reference Manual* for information on the `LDR` *instruction*.

Syntax

The syntax of `LDR` is:

```
LDR register, =[expression | label-expression]
```

where:

`register` is the register to be loaded. `LDR` can access the low registers (r0-r7) only.

`expression`

evaluates to a numeric constant:

- If the value of *`expression`* is within range of a `MOV` instruction, the assembler generates the instruction.
- If the value of *`expression`* is *not* within range of a `MOV` instruction, the assembler places the constant in a literal pool and generates a program-relative `LDR` instruction that reads the constant from the literal pool.

  The offset from the pc to the constant must be positive and less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

`label-expression`

is a program-relative or external expression. The assembler places the value of *`label-expression`* in a literal pool and generates a program-relative `LDR` instruction that loads the value from the literal pool.

The offset from the pc to the value in the literal pool must be positive and less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-72 for more information.

If *label-expression* is an external expression, or is not contained in the current area, the assembler places a linker relocation directive in the object file. The linker ensures that the correct address is generated at link time.

Usage

The LDR pseudo-instruction is used for two main purposes:

- to generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV instruction.
- to load a program-relative or external address into a register. The address remains valid regardless of where the linker places the AOF area containing the LDR.

Refer to Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for a more detailed explanation of how to use LDR, and for more information on MOV.

Example

```
        LDR     r1, =0xfff      ; loads 0xfff into r1
                                ;
        LDR     r2, =labelname  ; loads the address of
                                ;   labelname into r2
```

5.6.3 MOV Thumb pseudo-instruction

The Thumb `MOV` *pseudo*-instruction moves the value of a low register to another low register (r0-r7).

The Thumb `MOV` *instruction* cannot move values from one low register to another.

Note

The `ADD` immediate instruction generated by the assembler has the side-effect of updating the condition codes.

Syntax

The syntax of `MOV` is:

`MOV` *`Rd`*`,`*`Rs`*

where:

`Rd` is the destination register.

`Rs` is the source register.

Usage

The `MOV` pseudo-instruction uses an `ADD` immediate instruction with a zero immediate value.

Refer to the *ARM Architectural Reference Manual* for more information on the Thumb `MOV` instruction.

Example

```
        MOV Rd, Rs  ; generates the opcode for ADD Rd, Rs, #0
```

5.6.4 NOP Thumb pseudo-instruction

`NOP` generates the preferred Thumb no-operation instruction. This is:

```
MOV r8,r8
```

Syntax

The syntax for `NOP` is:

```
NOP
```

Usage

Condition codes are unaltered by `NOP`.

5.7 Symbols

You can use symbols to represent variables, addresses, and numeric constants. Symbols representing addresses are also called labels. See:

- *Variables* on page 5-26
- *Numeric constants* on page 5-29
- *Labels* on page 5-27.

5.7.1 Symbol naming rules

The following general rules apply to symbol names:

- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names.
- Do not use numeric characters for the first character of symbol names, except in local labels. See *Local labels* on page 5-28.
- Symbol names are case-sensitive.
- All characters in the symbol name are significant.
- Symbol names must be unique within their scope.
- Symbols must not use built-in variable names or predefined symbol names. See *Predefined register and coprocessor names* on page 5-9 and *Built-in variables* on page 5-10.
- Symbols should not use the same name as instruction mnemonics or directives. The assembler can distinguish between them through their relative positions on the input line but it makes the code difficult to read.

If you need to use a wider range of characters in symbols, for example, when working with compilers, use enclosing bars to delimit the symbol name. For example:

```
|C$$code|
```

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines even within the bars.

5.7.2 Variables

The value of a variable can be changed as assembly proceeds. Variables are of three types:

- numeric
- logical
- string.

The type of a variable cannot be changed.

The range of possible values of a numeric variable is the same as the range of possible values of a numeric constant or numeric expression. See *Numeric constants* on page 5-29 and *Numeric expressions* on page 5-89.

The range of possible values of a logical variable is `{TRUE}` or `{FALSE}`. See *Logical expressions* on page 5-90.

The range of possible values of a string variable is the same as the range of values of a string expression. See *String expressions* on page 5-88.

Use the `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, and `LCLS` directives to declare symbols representing variables, and assign values to them using the `SETA`, `SETL`, and `SETS` directives. See:

- *GBLA directive* on page 5-60
- *GBLL directive* on page 5-61
- *GBLS directive* on page 5-62
- *LCLA directive* on page 5-69
- *LCLL directive* on page 5-70
- *LCLS directive* on page 5-71
- *SETA directive* on page 5-82
- *SETL directive* on page 5-83
- *SETS directive* on page 5-84.

5.7.3 Assembly time substitution of variables

You can use a string variable for a whole line of assembly language, or any part of a line. Use the variable with a $ prefix in the places where the value is to be substituted for the variable. The dollar character instructs the assembler to substitute the string into the source code line before checking the syntax of the line.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name. See *Symbol naming rules* on page 5-25. You must set the contents of the variable before you can use it.

Example

```
add4ff  SETS    " ADD r4,r4,#0xFF"  ; set up add4ff
                                    ; (note the leading space)
        $add4ff.00                  ; invoke add4ff,
                                    ; producing
        ADD     r4,r4,#0xFF00
```

5.7.4 Labels

Labels are symbols representing the addresses in memory of instructions or data. They may be program-relative, register-relative, or absolute:

Program-relative labels

These represent the program counter plus or minus a numeric constant. Use them as targets for branch instructions, or to access small items of data embedded in code areas. You can define program-relative labels using a label on an instruction or on one of the Define Constant directives. See:

- *DCB or = directive* on page 5-46
- *DCD or & directive* on page 5-47
- *DCDU directive* on page 5-48
- *DCFD directive* on page 5-49
- *DCFDU directive* on page 5-50
- *DCFS directive* on page 5-51
- *DCFSU directive* on page 5-52
- *DCW directive* on page 5-53
- *DCWU directive* on page 5-54.

Register-relative labels

These represent a named register plus a numeric constant. They are most often used to access data in data areas. You can define them with any of the Define Constant directives, the `BASED Rn` attribute of the `AREA` directive, or with a storage map. See:

- ^ *or MAP directive* on page 5-35
- *% directive* on page 5-32
- *AREA directive* on page 5-38.

Absolute addresses

These are numeric constants. They are integers in the range 0 to 2^{32}–1. They address the memory directly. The most common uses of absolute addresses are in exception handling routines and for accessing memory-mapped I/O ports.

5.7.5 Local labels

A local label is a number in the range 0-99, optionally followed by a name. The same number can be used for more than one local label in an AOF area.

Local labels are used for instructions that are the target for branches. You cannot use them for data. Typically they are used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful in macros. See *MACRO directive* on page 5-73.

Use the `ROUT` directive to limit the scope of local labels. See *ROUT directive* on page 5-81. A reference to a local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, the assembler generates an error message and the assembly fails.

You can use the same number for more than one local label even within the same scope. By default, the assembler links a reference to a local label:

- to the most recent local label of the same number, if there is one within the scope
- to the next following local label of the same number, if there is not one within the scope.

Use the optional parameters to modify this search pattern if required.

Syntax

The syntax of a local label is:

```
n{routname}
```

The syntax of a reference to a local label is:

`%{F|B}{A|T}`*`n`*`{`*`routname`*`}`

where:

`n`	is the number of the local label.
`routname`	is the name of the current scope.
`%`	introduces the reference.
`F`	instructs the assembler to search forwards only.
`B`	instructs the assembler to search backwards only.
`A`	instructs the assembler to search all macro levels.
`T`	instructs the assembler to look at this macro level only.

If neither `F` or `B` is specified, the assembler searches backwards first, then forwards.

If neither `A` or `T` is specified, the assembler searches all macros from the current level to the top level, but does not search lower level macros.

If *`routname`* is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding `ROUT` directive. If it does not match, the assembler generates an error message and the assembly fails.

5.7.6 Numeric constants

Numeric constants are 32-bit integers. You can set them using unsigned numbers in the range 0 to $2^{32}–1$, or signed numbers in the range $–2^{31}$ to $2^{31}–1$. However, the assembler makes no distinction between –*n* and $2^{32}– n$. Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

Use the `EQU` directive to define constants. See *EQU or * directive* on page 5-57. You cannot change the value of a numeric constant after you define it.

5.8 Directives

The assembler provides directives to support:

- data structure definitions and allocation of space for data
- partitioning of files into logical subdivisions
- error reporting and control of assembly listing
- definition of symbols
- conditional and repetitive assembly, and inclusion of subsidiary files.

See Table 5-1 on page 5-2 to locate individual directives within this section. The directives are described in the following sections in alphabetical order.

5.8.1 Nesting directives

`MACRO` definitions, `WHILE...WEND` loops, `IF...ENDIF` conditions and `GET` or `INCLUDE` directives can be nested within themselves or within each other to a total depth of 256.

5.8.2 ! directive

See *INFO or ! directive* on page 5-67.

5.8.3 # directive

The # directive describes space within a storage map that has been defined using the ^ directive.

Syntax

The syntax of # is:

{*label*} # *expression*

where:

label is an optional label. If specified, *label* is assigned the value of the storage location counter, @. The storage location counter is then incremented by the value of *expression*.

expression

is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a ^ directive that specifies a *base-register*, the base register is implicit in all labels defined by following # directives, until the next ^ directive. These register-relative labels can be quoted in load and store instructions. Refer to ^ *or MAP directive* on page 5-35.

Note

You must be careful when using ^, #, and register-relative labels. Refer to Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for more information.

Example

The following example shows how register-relative labels are defined using the ^ and # directives.

```
        ^       0,r9        ; set @ to the address stored in r9
        #       4           ; increment @ by 4 bytes
Lab     #       4           ; set Lab to the address [r9 + 4]
                            ; and then increment @ by 4 bytes
        LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

5.8.4 % directive

The `%` directive reserves a zeroed block of memory.

Syntax

The syntax of `%` is:

`{`*`label`*`} %` *`numeric-expression`*

where:

`numeric-expression`

evaluates to the number of zeroed bytes to reserve.

Usage

You *must* use a `DATA` directive if you use `%` to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

Use the `ALIGN` directive to align any code following a `%` directive. Refer to *ALIGN directive* on page 5-36 for more information.

See also:

- *DCB or = directive* on page 5-46
- *DCW directive* on page 5-53
- *DCD or & directive* on page 5-47
- *DCWU directive* on page 5-54
- *DCDU directive* on page 5-48.

Example

```
        AREA    MyData, DATA, READWRITE
data1   %       255         ; defines 255 bytes of zeroed store
```

5.8.5 & directive

See *DCD or & directive* on page 5-47.

5.8.6 * directive

See *EQU or * directive* on page 5-57.

5.8.7 = directive

See *DCB or = directive* on page 5-46.

5.8.8 [or IF directive

The `IF` directive introduces a condition that is used to decide whether to assemble a sequence of instructions and/or directives. `[` and `IF` are synonyms.

Syntax

The syntax of `IF` is:

```
IF logical-expression

 ...

{ELSE

 ...}

ENDIF
```

where:

`logical-expression`
is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

See *Relational operators* on page 5-92.

Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition. See also *| or ELSE directive* on page 5-34 and *] or ENDIF directive* on page 5-34.

`IF...ENDIF` conditions can be nested. See *Nesting directives* on page 5-30.

Example

```
[ Version = "1.0"                ; IF ...
; code and/or
; directives
|                                ; ELSE
; code and/or
; directives
]                                ; ENDIF
```

5.8.9 | or ELSE directive

The `ELSE` directive marks the beginning of a sequence of instructions and/or directives that are to be assembled if the preceding condition fails. `|` and `ELSE` are synonyms.

Syntax

The syntax of `ELSE` is:

```
ELSE
```

Usage

See *[or IF directive* on page 5-33.

5.8.10] or ENDIF directive

The `ENDIF` directive marks the end of a sequence of instructions and/or directives that are to be conditionally assembled. `]` and `ENDIF` are synonyms.

Syntax

The syntax of `ENDIF` is:

```
ENDIF
```

Usage

See *[or IF directive* on page 5-33.

5.8.11 ^ or MAP directive

The `^` directive sets the origin of a storage map to a specified address. The storage-map location counter, @, is set to the same address. `MAP` is a synonym for `^`.

Syntax

The syntax of `^` is:

`^` *`expression`*`{,`*`base-register`*`}`

where:

`expression`

is a numeric or program-relative expression:

- If *`base-register`* is not specified, *`expression`* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If the expression is program-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

`base-register`

specifies a register. If *`base-register`* is specified, the address where the storage map starts is the sum of *`expression`*, and the value in *`base-register`* at runtime.

Usage

Use the `^` directive in combination with the `#` directive to describe a storage map.

Specify *`base-register`* to define register-relative labels. The base register becomes implicit in all labels defined by following `#` directives, until the next `^` directive. The register-relative labels can be used in load and store instructions. Refer to *# directive* on page 5-31 for an example.

The `^` directive can be used any number of times to define multiple storage maps.

The @ counter is set to zero before the first `^` directive is used.

Examples

```
        ^       0,r3
        ^       0xff,r3
```

5.8.12 ALIGN directive

By default, the `ALIGN` directive aligns the current location within the code to a word (4-byte) boundary.

Syntax

The syntax of `ALIGN` is:

`ALIGN {`*`expression`*`{,`*`offset-expression`*`}}`

where:

`expression`

can be any power of 2 from 2^0 to 2^{31}. The current location is aligned to the next 2^n-byte boundary. If this parameter is not specified, `ALIGN` sets the instruction location to the next word boundary.

`offset-expression`

defines a byte offset from the alignment specified by *`expression`*.

Usage

Use `ALIGN` to ensure that your code is correctly aligned. As a general rule it is safer to use `ALIGN` frequently through your code.

Use `ALIGN` to ensure that Thumb addresses are word aligned when required. For example, the `ADR` Thumb pseudo-instruction can only load addresses that are word aligned.

Use `ALIGN` when data definition directives appear in code areas. When data definition directives (`DCB`, `DCW`, `DCWU`, `DCDU` and `%`) are used in code areas, the program counter does not necessarily point to a word boundary. When the assembler encounters the next instruction mnemonic it inserts up to 3 bytes, if required, to ensure that the instruction is:

- word aligned in ARM state
- halfword aligned in Thumb state.

In this case, a label that appears on a source line by itself does not address the following instruction. Use `ALIGN` to ensure that the label addresses the following instruction. You can use `ALIGN 2` to align on a halfword (2-byte) boundary in Thumb code.

Use `ALIGN` with a coarser setting to take advantage of caches on some ARM processors. For example, the ARM940T has a cache with 4-word lines. Use `ALIGN 16` to align function entries on 16-byte boundaries and maximize the efficiency of the cache.

Alignment is relative to the start of the AOF area where the routine is located. You must ensure that the area is also aligned to the same, or coarser, boundaries. The `ALIGN` attribute on the `AREA` directive is specified differently. See *AREA directive* on page 5-38 and the example below.

Examples

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        DCB     1           ; pc misaligned
        ALIGN               ; ensures that label1 addresses
label1                      ; the following instruction.
        MOV r5,#0x5


        AREA    cacheable, CODE, ALIGN=4
rout1   ; code              ; aligned on 16-byte boundary
        ; code
        MOV     pc,lr       ; aligned only on 4-byte boundary
        ALIGN   16          ; now aligned on 16-byte boundary
rout2   ; code
```

5.8.13 AREA directive

The `AREA` directive instructs the assembler to assemble a new code or data area. Areas are independent, named, indivisible chunks of code or data that are manipulated by the linker. Refer to Chapter 5 *Basic Assembly Language Programming* and Chapter 6 *Linker* in the *ARM Software Development Toolkit Reference Guide* for more information.

Syntax

The syntax of the `AREA` directive is:

```
AREA name{,attr}{,attr}...
```

where:

`name` is the name that the area is to be given.

You can choose any name for your areas. However, names starting with a digit must be enclosed in bars or a missing area name error is generated. For example, `|1_DataArea|`.

Certain names are conventional. For example, `|C$$code|` is used for code areas produced by the C compiler, or for code areas otherwise associated with the C library.

`attr` are one or more comma-delimited area attributes. Valid attributes are:

`ALIGN=`*`expression`*

By default, AOF areas are aligned on a 4-byte boundary.

`expression` can have any integer value between 2 and 31. The area is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the area is aligned on a 1KB boundary.

`CODE` Contains machine instructions. `READONLY` is the default.

`COMDEF` Is a common area definition. This AOF area may contain code or data. It must be identical to any other area of the same name in other source files.

Identical AOF areas with the same name are overlaid in the same area of memory by the linker. If any are different, the linker generates a warning and does not overlay the areas.

`COMMON` Is a common data area. You must not define any code or data in it. It is initialized to zeroes by the linker. All common areas with the same name are overlaid in the same area of memory by the linker. They do not all need to be the same size. The linker allocates as much space as is required by the largest common area of each name.

`DATA` Contains data, not instructions. `READWRITE` is the default.

`INTERWORK`

Indicates that the code area is suitable for ARM/Thumb interworking.

`NOINIT` Indicates that the data area is initialized to zero. It contains only space reservation directives, with no initialized values.

`PIC` Indicates position-independent code. It can execute at any address without modification.

`READONLY`

Indicates that this area should not be written to.

`READWRITE`

Indicates that this area may be read from and written to.

Usage

There must be at least one `AREA` directive for an assembly.

Use the `AREA` directive to subdivide your source file into AOF areas. You must use a different name for each area within the same source file.

You should normally use separate AOF areas for code and data. Large programs can usually be conveniently divided into several code areas. Large independent data sets are also usually best placed in separate areas.

The scope of local labels is defined by AOF areas, optionally subdivided by `ROUT` directives. See *Local labels* on page 5-28 and *ROUT directive* on page 5-81.

If no `AREA` directive is specified, the assembler generates an AOF area with the name `|$$$$$$$|`, and produces a diagnostic message. This limits the number of error messages caused by the missing directive, but does not lead to a successful assembly.

Example

The following example defines a read-only code area named Example.

```
        AREA    Example,CODE,READONLY   ; An example code area.
        ; code
```

5.8.14 ASSERT directive

The ASSERT directive generates an error message during the second pass of the assembly if a given assertion is false.

Syntax

The syntax of ASSERT is:

ASSERT *logical-expression*

where:

logical-expression

is an assertion that can evaluate to either {TRUE} or {FALSE}.

Usage

Use ASSERT to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

See also *INFO or ! directive* on page 5-67.

Example

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

5.8.15 CN directive

The `CN` directive defines a name for a coprocessor register.

Syntax

The syntax of `CN` is:

`name` `CN` *`numeric-expression`*

where:

`name` is the name to be defined for the coprocessor register.

`numeric-expression`
evaluates to a coprocessor register number from 0 to 15.

Usage

Use `CN` to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power       CN  6       ; defines power as a symbol for
                        ; coprocessor register 6
```

5.8.16 CODE16 directive

The CODE16 directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions.

Syntax

The syntax of CODE16 is:

```
CODE16
```

Usage

Use CODE16 when branching to Thumb state with the BX instruction. CODE16 precedes code at the destination of the branch.

The assembler inserts a byte of padding, if necessary, to bring following Thumb code into halfword alignment. CODE16 does not assemble to an instruction that changes the mode. It only instructs the assembler to assemble Thumb instructions.

See also *CODE32 directive* on page 5-43.

Example

This example shows how CODE16 can be used to branch from ARM to Thumb instructions.

```
        AREA    ThumbEx, CODE, READONLY

                            ; This area starts in ARM state
        ADR     r0,start+1  ; Load the address and set the
                            ; least significant bit
        BX      r0          ; Branch and exchange
                            ; instruction sets

                            ; Not necessarily in the same area
        CODE16              ; Following instructions are Thumb
start   MOV     r1,#10      ; Thumb instructions
```

5.8.17 CODE32 directive

The `CODE32` directive instructs the assembler to interpret subsequent instructions as 32-bit ARM instructions.

Syntax

The syntax of `CODE32` is:

```
CODE32
```

Usage

Use `CODE32` to when branching to ARM state from Thumb state. `CODE32` precedes code at the destination of the branch.

The assembler inserts up to three bytes of padding, if necessary, to bring following ARM code into word alignment. `CODE32` does not assemble to an instruction that changes the mode. It only instructs the assembler to assemble ARM instructions.

See also *CODE16 directive* on page 5-42.

Example

```
        CODE16              ; Start this area in Thumb state

        AREA    ThumbEx, CODE, READONLY

        MOV     r1,#10      ; Thumb instructions
        ADR     r0,goarm    ; Load the address and leave the
                            ; least significant bit clear.
        BX      r0          ; Branch and exchange instruction
                            ; sets

                            ; Not necessarily in the same area
        CODE32              ; Following instructions are ARM
goarm   MOV     r4,#15      ; ARM instructions
```

5.8.18 CP directive

The `CP` directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

Syntax

The syntax of `CP` is:

`name` `CP` *`numeric-expression`*

where:

`name` is the name to be assigned to the coprocessor. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-9.

`numeric-expression`

evaluates to a coprocessor number from 0 to 15.

Usage

Use `CP` to allocate convenient names to coprocessors, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu         CP  6       ; defines dmu as a symbol for
                        ; coprocessor 6
```

5.8.19 DATA directive

The `DATA` directive informs the assembler that a label is a *data-in-code* label. This means that the label is the address of data within a code segment.

Syntax

The syntax of `DATA` is:

```
label DATA
```

where:

`label` is the label of the data definition. The `DATA` directive must be on the same line as *`label`*.

Usage

You *must* use the `DATA` directive when you define data in a Thumb code area with any of the data-defining directives such as `DCD`, `DCB`, and `DCW`.

When the linker relocates a label in a Thumb code area, it assumes that the label represents the address of a Thumb routine. The linker adds 1 to the value of the label so that the processor is switched to Thumb state if the routine is called with a `BX` instruction.

If a label represents the address of data within a Thumb code area, you do not want the linker to add 1 to the label. The `DATA` directive marks the label as pointing to data within a code area and the linker does not add 1 to its value.

You can use `DATA` to mark data-in-code in ARM code areas. The `DATA` directive is ignored by the assembler in ARM code areas.

Example

```
            AREA    example, CODE
Thumb_fn    ; code
            ; code
            MOV     pc, lr

Thumb_Data  DATA
            DCB     1, 3, 4
```

5.8.20 DCB or = directive

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.

Syntax

The syntax of DCB is:

```
{label} DCB expression{,expression}...
```

where:

expression

is either:

- A numeric expression that evaluates to an integer in the range –128 to 255. See *Numeric expressions* on page 5-89.
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

You *must* use the DATA directive if you use DCB to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

If DCB is followed by an instruction, use an ALIGN directive to ensure that the instruction is aligned. Refer to *ALIGN directive* on page 5-36 for more information.

See also:

- *DCW directive* on page 5-53
- *DCD or & directive* on page 5-47
- *DCWU directive* on page 5-54
- *DCDU directive* on page 5-48
- *% directive* on page 5-32

Example

Unlike C strings, ARM assembler strings are not null-terminated. You can construct a null-terminated C string using DCB as follows:

```
C_string     DCB    "C_string",0
```

5.8.21 DCD or & directive

The `DCD` directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory. `&` is a synonym for `DCD`.

Syntax

The syntax of `DCD` is:

`{`*`label`*`} DCD `*`expression`*`{,`*`expression`*`}`

where:

`expression`

is either:

- A numeric expression. See *Numeric expressions* on page 5-89.
- A program-relative expression.

Usage

You *must* use the `DATA` directive if you use `DCD` to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

`DCD` inserts up to 3 bytes of padding before the first defined word, if necessary, to achieve 4-byte alignment. Use `DCDU` if you do not require alignment.

See also:

- *DCB or = directive* on page 5-46
- *DCW directive* on page 5-53
- *DCWU directive* on page 5-54
- *DCDU directive* on page 5-48
- *% directive* on page 5-32

Example

```
data1   DCD     1,5,20          ; Defines 3 words containing
                                ; decimal values 1, 5, and 20

data2   DCD     mem06           ; Defines 1 word containing the
                                ; address of the label mem06

data3   DCD     glb + 4         ; Defines 1 word containing
                                ; 4 + the value of glb
```

5.8.22 DCDU directive

The `DCDU` directive allocates one or more words of memory, not necessarily aligned, and defines the initial runtime contents of the memory.

Syntax

The syntax of `DCDU` is:

{*label*} DCDU *expression*{,*expression*}...

where:

expression

is either:

- A numeric expression. See *Numeric expressions* on page 5-89.
- A program-relative expression.

Usage

Use `DCDU` to define data words with arbitrary alignment.

You *must* use the `DATA` directive if you use `DCDU` to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

If `DCDU` is followed by code, use an `ALIGN` directive to ensure that the instructions are word aligned. Refer to *ALIGN directive* on page 5-36 for more information.

`DCDU` does not insert padding when preceding code is unaligned. Use `DCD` if you require alignment.

See also:

- *DCB or = directive* on page 5-46
- *DCW directive* on page 5-53
- *DCD or & directive* on page 5-47
- *DCWU directive* on page 5-54
- *% directive* on page 5-32

Example

```
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data1   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

5.8.23 DCFD directive

The `DCFD` directive allocates memory for word-aligned double precision floating-point numbers, and defines the initial runtime contents of the memory. Double precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

Syntax

The syntax of `DCFD` is:

```
{label} DCFD fp-constant{,fp-constant}...
```

where:

`fp-constant`

is a double precision floating-point value in one of the following forms:

```
{-}digits E{-}digits
{-}{digits}.digits{E{-}digits}
```

`E` may also be written in lowercase.

Usage

The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve 4-byte alignment. Use `DCFDU` if you do not require alignment.

The range for double precision numbers is:

- Maximum 1.79769313486231571e+308
- Minimum 2.22507385850720138e–308

Examples

```
        DCFD    1E308,-4E-100
        DCFD    10000,-.1,3.1E26
```

5.8.24 DCFDU directive

The `DCFDU` directive allocates eight bytes of memory for arbitrarily aligned double precision floating-point numbers, and defines the initial runtime contents of the memory.

Syntax

The syntax of `DCFDU` is:

{*label*} DCFDU *fp-constant*{,*fp-constant*}...

where:

fp-constant

is a double precision floating-point value in one of the following forms:

{-}*digits* E{-}*digits*
{-}{*digits*}.*digits*{E{-}*digits*}

`E` may also be written in lowercase.

Usage

`DCFDU` defines floating-point values with arbitrary alignment.

The range for double precision numbers is:

- Maximum 1.79769313486231571e+308
- Minimum 2.22507385850720138e–308

Examples

```
        DCFDU   1E308,-4E-100
        DCFDU   100,-.1,3.1E26
```

5.8.25 DCFS directive

The `DCFS` directive allocates memory for word-aligned single precision floating-point numbers, and defines the initial runtime contents of the memory. Single precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

Syntax

The syntax of `DCFS` is:

```
{label} DCFS fp-constant{,fp-constant}...
```

where:

`fp-constant`

is a single precision floating-point value in one of the following forms:

```
{-}digits E{-}digits
{-}{digits}.digits{E{-}digits}
```

`E` may also be written in lowercase.

Usage

`DCFS` inserts up to three bytes of padding before the first defined number, if necessary to achieve 4-byte alignment. Use `DCFSU` if you do not require alignment.

The range for single precision values is:

- Maximum 3.40282347e+38F
- Minimum 1.17549435e–38F

Example

```
        DCFS    1E3,-4E-9
        DCFS    1.0,-.1,3.1E6
```

5.8.26 DCFSU directive

The `DCFSU` directive allocates memory for arbitrarily aligned single precision floating-point numbers, and defines the initial runtime contents of the memory.

Syntax

The syntax of `DCFSU` is:

{*label*} `DCFSU` *fp-constant*{,*fp-constant*}...

where:

fp-constant

is a single precision floating-point value in one of the following forms:

{-}*digits* E{-}*digits*
{-}{*digits*}.*digits*{E{-}*digits*}

`E` may also be written in lowercase.

Usage

Use `DCFSU` to define floating-point values with arbitrary alignment.

`DCFSU` does not insert padding when preceding code is unaligned. Use `DCFS` if you require alignment.

The range for single precision values is:

- Maximum 3.40282347e+38F
- Minimum 1.17549435e–38F

Example

```
        DCFSU   1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

5.8.27 DCW directive

The `DCW` directive allocates one or more halfwords of memory, aligned on 2-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

The syntax of `DCW` is:

{*label*} `DCW` *expression*{,*expression*}...

where:

expression

is a numeric expression that evaluates to an integer in the range –32768 to 65535. See *Numeric expressions* on page 5-89.

Usage

You *must* use a `DATA` directive if you use `DCW` to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

If `DCW` is followed by an instruction, use an `ALIGN` directive to ensure that the instruction is word aligned. Refer to *ALIGN directive* on page 5-36 for more information.

`DCW` inserts a byte of padding before the first defined halfword if necessary to achieve 2-byte alignment. Use `DCWU` if you do not require alignment.

See also:

- *DCB or = directive* on page 5-46
- *DCD or & directive* on page 5-47
- *DCWU directive* on page 5-54
- *DCDU directive* on page 5-48
- *% directive* on page 5-32

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCW     -225,2*number       ; number must already be
        DCW     number+4            ; defined
```

5.8.28 DCWU directive

The DCWU directive allocates one or more unaligned halfwords of memory, and defines the initial runtime contents of the memory.

Syntax

The syntax of DCWU is:

```
{label} DCWU expression{,expression}...
```

where:

expression

is a numeric expression that evaluates to an integer in the range –32768 to 65535. See *Numeric expressions* on page 5-89.

Usage

Use DCWU to define data halfwords with arbitrary alignment, in packed structures for example.

You *must* use a DATA directive if you use DCWU to define labeled data within Thumb code. Refer to *DATA directive* on page 5-45 for more information.

If DCWU is followed by code, use an ALIGN directive to ensure that instructions are word aligned. Refer to *ALIGN directive* on page 5-36 for more information.

DCWU does not insert padding when preceding code is unaligned. Use DCW if you require alignment.

See also:

- *DCB or = directive* on page 5-46
- *DCW directive* on page 5-53
- *DCD or & directive* on page 5-47
- *DCDU directive* on page 5-48
- *% directive* on page 5-32

Example

```
        AREA    DataB2, DATA, READWRITE
oddbits DCB     1,2,3           ; now not word aligned
        DCWU    number,-255,4   ; these will each occupy two
                                ; bytes, but not necessarily
                                ; aligned
```

5.8.29 ELSE directive

See *| or ELSE directive* on page 5-34

5.8.30 END directive

The `END` directive informs the assembler that it has reached the end of a source file.

Syntax

The syntax of `END` is:

```
END
```

Usage

Every assembly language source file must end with `END` on a line by itself.

If the source file has been included in a parent file by a `GET` directive, the assembler returns to the parent file and continues assembly at the first line following the `GET` directive. See *GET or INCLUDE directive* on page 5-63 for more information.

If `END` is reached in the top-level source file during the first pass without any errors, the second pass begins.

If `END` is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

5.8.31 ENDIF directive

See *] or ENDIF directive* on page 5-34

5.8.32 ENTRY directive

The ENTRY directive declares its offset in its containing AOF area to be the unique entry point to any program containing the area.

Syntax

The syntax of ENTRY is:

```
ENTRY
```

Usage

You must specify one and only one ENTRY directive for a program. If ENTRY does not exist, or if more than one ENTRY exists, a error message is generated at link time. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

For applications written entirely or partially in C or C++, the entry point is frequently located in the library code.

Example

```
        AREA    ARMex, CODE, READONLY
        ENTRY               ; Entry point for the application
```

5.8.33 EQU or * directive

The `EQU` directive gives a symbolic name to a numeric constant. `*` is a synonym for `EQU`.

Syntax

The syntax of `EQU` is:

```
name EQU expression
```

where:

`name` is the symbolic name to assign to the value.

`expression`
is a fixed, register-relative, or program-relative value.

Usage

Use `EQU` to define constants. This is similar to the use of **`#define`** to define a constant in C.

See also:

- *^ or MAP directive* on page 5-35
- *# directive* on page 5-31
- *Symbols* on page 5-25
- *Register-relative and program-relative expressions* on page 5-89.

Example

```
num     EQU     2       ; assigns the value 2 to the symbol num.
```

5.8.34 EXPORT or GLOBAL directive

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

Syntax

The syntax of `EXPORT` is:

`EXPORT` *`symbol`* `{[`*`qualifier`*`{,`*`qualifier`*`}{,`*`qualifier`*`}]}`

where:

`symbol` is the symbol name to export. The symbol name is case-sensitive.

`qualifier` can be any of:

`FPREGARGS`
meaning that *`symbol`* refers to a function that expects floating-point arguments to be passed in floating-point registers.

`DATA` meaning that *`symbol`* refers to a data location rather than a function or a procedure entry point.

`LEAF` denotes that the exported function is a leaf function that calls no other functions. This qualifier is obsolete.

Usage

Use `EXPORT` to allow code in other files to refer to symbols in the current file.

Use the `DATA` attribute to inform the linker that *`symbol`* should not be the target of branches.

See also *IMPORT or EXTERN directive* on page 5-64.

Example

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```

5.8.35 EXTERN directive

See *IMPORT or EXTERN directive* on page 5-64

5.8.36 FN directive

The `FN` directive defines a name for a specified floating-point register. The names f0-f7 and F0-F7 are predefined.

Syntax

The syntax of `FN` is:

```
name FN numeric-expression
```

where:

`name` is the name to be assigned to the floating-point register. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-9.

`numeric-expression`
evaluates to a floating-point register number from 0 to 7.

Usage

Use `FN` to allocate convenient names to floating-point registers, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same register under different names.

Example

```
energy      FN  6           ; defines energy as a symbol for
                            ; floating-point register 6
```

5.8.37 GBLA directive

The `GBLA` directive declares and initializes a global arithmetic variable. The range of values that arithmetic variables may take is the same as that of numeric expressions. See *Numeric expressions* on page 5-89.

Syntax

The syntax of `GBLA` is:

`GBLA` *`variable-name`*

where:

`variable-name`

is the name of the arithmetic variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to 0.

Usage

Using `GBLA` for a variable that is already defined re-initializes the variable to 0. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETA` directive. See *SETA directive* on page 5-82.

See also *LCLA directive* on page 5-69 for information on setting local arithmetic variables.

Global variables can also be set with the `-predefine` assembler command-line option. Refer to *Command syntax* on page 5-3 for more information.

Example

```
            GBLA    objectsize      ; declare the variable name
objectsize  SETA    0xff            ; set its value
            %       objectsize      ; quote the variable
```

5.8.38 GBLL directive

The `GBLL` directive declares and initializes a global logical variable. Possible values of a logical variable are `{TRUE}` and `{FALSE}`.

Syntax

The syntax of `GBLL` is:

`GBLL` *`variable-name`*

where:

`variable-name`

is the name of the logical variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to `{FALSE}`.

Usage

Using `GBLL` for a variable that is already defined re-initializes the variable to `{FALSE}`. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETL` directive. See *SETL directive* on page 5-83.

See *LCLL directive* on page 5-70 for information on setting local logical variables.

Global variables can also be set with the `-predefine` assembler command-line option. Refer to *Command syntax* on page 5-3 for more information.

Example

```
        GBLL    testrun
testrun SETL    {TRUE}

        IF      testrun
        ; testcode
        ENDIF
```

5.8.39 GBLS directive

The `GBLS` directive declares and initializes a global string variable. The range of values that string variables may take is the same as that of string expressions. See *String expressions* on page 5-88.

Syntax

The syntax of `GBLS` is:

`GBLS` *`variable-name`*

where:

`variable-name`

is the name of the string variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to a null string, `""`.

Usage

Using `GBLS` for a variable that is already defined re-initializes the variable to a null string. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETS` directive. See *SETS directive* on page 5-84.

See *LCLS directive* on page 5-71 for information on setting local string variables.

Global variables can also be set with the `-predefine` assembler command-line option. Refer to *Command syntax* on page 5-3 for more information.

Example

```
        GBLS    version          ; declare the variable
version SETS    "Version 1.0"    ; set its value
        ; code
        INFO    0,version        ; use the variable
```

5.8.40 GET or INCLUDE directive

The GET directive includes a file within the file being assembled. The included file is assembled. INCLUDE is a synonym for GET.

Syntax

The syntax of GET is:

GET *filename*

where:

filename is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

GET is useful for including macro definitions, EQUs, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the GET directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names must not contain spaces.

The included file may contain additional GET directives to include other files. See *Nesting directives* on page 5-30.

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

GET cannot be used to include object files. See *INCBIN directive* on page 5-66.

Example

```
    AREA    Example, CODE, READONLY
    GET     file1.s             ; includes file1 if it exists
                                ; in the current place.
    GET     c:\project\file2.s  ; includes file2
```

5.8.41 GLOBAL directive

See *EXPORT or GLOBAL directive* on page 5-58

5.8.42 IF directive

See *[or IF directive* on page 5-33

5.8.43 IMPORT or EXTERN directive

The `IMPORT` directive provides the assembler with a name that is not defined in the current assembly. `EXTERN` is a synonym for `IMPORT`. See also *EXPORT or GLOBAL directive* on page 5-58.

Syntax

The syntax of `IMPORT` is:

```
IMPORT symbol{[qualifier{,qualifier}]}
```

where:

`symbol` is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

`qualifier` can be:

`FPREGARGS`
specifies that *`symbol`* defines a function that expects floating-point arguments passed in floating-point registers.

`WEAK` prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- if the reference is the destination of a Branch or Branch Link instruction, the value of the symbol is taken as the address of the referencing instruction. The instruction becomes `B {PC}` or `BL {PC}`.

- otherwise, the value of the symbol is taken as zero.

You must avoid executing `B {PC}` and `BL {PC}` at runtime as they are non-terminating loops.

To avoid trying to access symbols that are not found at link time, use code like the example below to test your environment at runtime.

Example

This example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
        AREA    Example, CODE, READONLY
        IMPORT  __CPP_INITIALIZE[WEAK]  ; If C++ library linked
                                        ; gets the address of
                                        ; CPP_INIT function.
        LDR     r0,__CPP_INITIALIZE     ; If not linked, address
                                        ; is zeroed.
        CMP     r0,#0                   ; Test if zero.
        BEQ     nocplusplus             ; Branch on the result.
```

5.8.44 INCBIN directive

The `INCBIN` directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

The syntax of `INCBIN` is:

`INCBIN` *`filename`*

where:

`filename` is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

You can use `INCBIN` to include executable files, literals, or any arbitrary data. The contents of the file are added to the current AOF area, byte for byte, without being interpreted in any way. Assembly continues at the line following the `INCBIN` directive.

By default the assembler searches the current place for included files. See *GET or INCLUDE directive* on page 5-63 for information on the current place. Use the `-i` assembler command-line option to add directories to the search path. File names and directory names must not contain spaces.

Example

```
        AREA    Example, CODE, READONLY
        INCBIN  file1.dat               ; includes file1 if it
                                        ; exists in the
                                        ; current place.
        INCBIN  c:\project\file2.txt    ; includes file2
```

5.8.45 INCLUDE directive

See *GET or INCLUDE directive* on page 5-63

5.8.46 INFO or ! directive

The `INFO` directive supports diagnostic generation on either pass of the assembly.

! is a synonym for `INFO`.

Syntax

The syntax of `INFO` is:

`INFO` *`numeric-expression, string-expression`*

where:

`numeric-expression`

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- no action is taken during pass one
- *`string-expression`* is printed during pass two.

If the expression does not evaluate to zero, *`string-expression`* is printed as an error message and the assembly fails.

`string-expression`

is an expression that evaluates to a string.

Usage

`INFO` provides a flexible means for creating custom error messages. See *Numeric expressions* on page 5-89 and *String expressions* on page 5-88 for additional information on numeric and string expressions.

See also *ASSERT directive* on page 5-40.

Examples

```
INFO    0, "Version 1.0"

IF endofdata <= label1
INFO    4, "Data overrun at label1"
ENDIF
```

5.8.47 KEEP directive

The `KEEP` directive instructs the assembler to retain local symbols in the symbol table in the object file.

Syntax

The syntax of `KEEP` is:

`KEEP {`*`symbol`*`}`

where:

`symbol` is the name of the local symbol to keep. If *`symbol`* is not specified, all local symbols are kept except register-relative symbols.

Usage

By default, the assembler only describes exported symbols in its output object file. Use `KEEP` to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

`KEEP` can be used only after the definition of *`symbol`*. This does not apply to `KEEP` without *`symbol`*.

`KEEP` cannot preserve register-relative symbols. See *^ or MAP directive* on page 5-35.

Example

```
label   ADC     r2,r3,r4
        KEEP    label        ; makes label available to debuggers
        ADD     r2,r2,r5
```

5.8.48 LCLA directive

The `LCLA` directive declares and initializes a local arithmetic variable. Local variables can be declared only within a macro.

The range of values that arithmetic variables may take is the same as that of numeric expressions. See *Numeric expressions* on page 5-89.

Syntax

The syntax of `LCLA` is:

`LCLA` *`variable-name`*

where:

`variable-name`

is the name of the variable to set. The name must be unique within the macro that contains it. The initial value of the variable is 0.

Usage

See also *MACRO directive* on page 5-73. The scope of the variable is limited to a particular instantiation of the macro that contains it.

Using `LCLA` for a variable that is already defined re-initializes the variable to 0.

Set the value of the variable with the `SETA` directive. See *SETA directive* on page 5-82.

See *GBLA directive* on page 5-60 for information on declaring global arithmetic variables.

Example

```
                                ; Calculate the next-power-of-2
                                ; number >= the value given.
        MACRO                   ; Declare a macro
$rslt   NPOW2   $value          ; Macro prototype line
        LCLA    newval          ; Declare local arithmetic
                                ; variable newval.
newval  SETA    1               ; Set value of newval to 1
        WHILE   (newval < $value)
                                ; Repeat a loop that
newval  SETA    (newval :SHL:1) ; multiplies newval by 2
        WEND                    ; until newval >= $value.
$rslt   EQU     (newval)        ; Return newval in $rslt
        MEND                    ; No runtime instructions here!
```

5.8.49 LCLL directive

The `LCLL` directive declares and initializes a local logical variable. Local variables can be declared only within a macro. Possible values of a logical variable are `{TRUE}` and `{FALSE}`.

Syntax

The syntax of `LCLL` is:

`LCLL` *`variable-name`*

where:

`variable-name`

is the name of the variable to set. The name must be unique within the macro that contains it. The initial value of the variable is `{FALSE}`.

Usage

See also *MACRO directive* on page 5-73. The scope of the variable is limited to a particular instantiation of the macro that contains it.

Using `LCLL` for a variable that is already defined re-initializes the variable to `{FALSE}`.

Set the value of the variable with the `SETL` directive. See *SETL directive* on page 5-83.

See *GBLL directive* on page 5-61 for information on declaring global logical variables.

Example

```
        MACRO                   ; Declare a macro
$label  cases   $x              ; Macro prototype line
        LCLL    xisodd          ; Declare local logical variable
                                ; xisodd.
xisodd  SETL    $x:MOD:2=1      ; Set value of xisodd according
                                ; to $x
$label  ; code
        IF      xisodd          ; Assemble following code only
                                ; if $x is odd.
        ; code
        ENDIF
        MEND                    ; End of macro
```

5.8.50 LCLS directive

The `LCLS` directive declares and initializes a local string variable. Local variables can be declared only within a macro. The initial value of the variable is a null string, `""`.

Syntax

The syntax of `LCLS` is:

`LCLS` *`variable-name`*

where:

`variable-name`

is the name of the variable to set. The name must be unique within the macro that contains it.

Usage

See also *MACRO directive* on page 5-73. The scope of the variable is limited to a particular instantiation of the macro that contains it.

Using `LCLS` for a variable that is already defined re-initializes the variable to a null string.

Set the value of the variable with the `SETS` directive. See *SETS directive* on page 5-84.

See *GBLS directive* on page 5-62 for information on declaring global logical variables.

Example

```
        MACRO                      ; Declare a macro
$label  message $a                 ; Macro prototype line
        LCLS    err                ; Declare local string
                                   ; variable err.
err     SETS    "error no: "       ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a ; Use string
        MEND
```

5.8.51 LTORG directive

The LTORG directive instructs the assembler to assemble the current literal pool immediately.

Syntax

The syntax of LTORG is:

```
LTORG
```

Usage

The assembler executes an LTORG directive at the end of every code area, as defined by the AREA directive at the beginning of the following area, or the end of the assembly.

Use LTORG to ensure that literal pools are assembled within range of the LDR, LDFD, and LDFS pseudo-instructions. Refer to *LDR ARM pseudo-instruction* on page 5-16 and *LDR Thumb pseudo-instruction* on page 5-21 for more information. Large programs may require several literal pools.

Place LTORG directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1

func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to
                                ; Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains
                                ; literal &55555555.

data    %       4200            ; Clears 4200 bytes of memory,
                                ; starting at current location.
        END                     ; Default literal pool is empty.
```

5.8.52 MACRO directive

The `MACRO` directive marks the start of the definition of a macro. Macro expansion terminates at the `MEND` directive. See Chapter 5 *Basic Assembly Language Programming* in the *ARM Software Development Toolkit User Guide* for further information.

Syntax

Two directives are used to define a macro. The syntax is:

```
    MACRO
macro_prototype
    ; code
    MEND
```

The `MACRO` directive must be followed by a macro prototype statement on the next line. The syntax of the macro prototype statement is:

`{$label} macroname {$parameter1{,$parameter2}...}`

where:

`$label` is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually, but not necessarily, a label.

`macroname` is the name of the macro. It must not begin with an instruction or directive name.

`$parameter`

is a parameter that is substituted when the macro is invoked. A default value for a parameter may be set using this format:

`$parameter="default value"`

Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

There must be no unclosed `WHILE...WEND` loops or unclosed `IF...ENDIF` conditions when the `MEND` directive is reached. See *MEXIT directive* on page 5-75 if you need to allow an early exit from a macro, for example from within a loop.

Within the macro body, parameters such as `$label`, `$parameter` can be used in the same way as other variables. See *Assembly time substitution of variables* on page 5-27. They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

`$label` is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

Macros define the scope of local variables. See *LCLA directive* on page 5-69, *LCLL directive* on page 5-70 and *LCLS directive* on page 5-71.

Macros can be nested. See *Nesting directives* on page 5-30.

Example

```
                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition

abc             xmac    subr1,de        ; invoke macro

                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

5.8.53 MAP directive

See ^ *or MAP directive* on page 5-35

5.8.54 MEND directive

The `MEND` directive marks the end of a macro definition.

See *MACRO directive* on page 5-73.

5.8.55 MEXIT directive

The `MEXIT` directive is used to exit a macro definition before the end.

Syntax

The syntax of `MEXIT` is:

```
MEXIT
```

Usage

Use `MEXIT` when you need an exit from within the body of a macro. Any unclosed `WHILE...WEND` loops or `IF...ENDIF` conditions are closed by the assembler before the macro is exited.

See also *MACRO directive* on page 5-73.

Example

```
        MACRO
$abc    macroabc        $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

5.8.56 NOFP directive

The `NOFP` directive disallows floating-point instructions in an assembly language source file.

Syntax

The syntax of `NOFP` is:

```
NOFP
```

Usage

Use `NOFP` to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software, or in target hardware.

If a floating-point instruction occurs after the `NOFP` directive, an `Unknown opcode` error is generated and the assembly fails.

If a `NOFP` directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

5.8.57 OPT directive

The `OPT` directive sets listing options from within the source code.

Syntax

The syntax of `OPT` is:

`OPT` *n*

where:

n is the `OPT` directive setting. Table 5-3 lists valid settings.

Table 5-3 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for `SET`, `GBL` and `LCL` directives.
32	Turns off listing for `SET`, `GBL` and `LCL` directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of `MEND` directives.
32768	Turns off listing of `MEND` directives.

Usage

Specify the `-list` assembler option to turn on listing.

By default the `-list` option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and `MEND` directives. The listing is produced on the second pass only. Use the `OPT` directive to modify the default listing options from within your code. Refer to *Command syntax* on page 5-3 for information on the `-list` option.

You can use `OPT` to format code listings. For example, you can specify a new page before functions and areas.

Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```

5.8.58 RLIST directive

The RLIST (register list) directive gives a name to a set of registers.

Syntax

The syntax of RLIST is:

```
name RLIST {list-of-registers}
```

where:

name is the name to be given to the set of registers.

list-of-registers
is a comma-delimited list of register names and/or register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the -checkreglist assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

```
Context RLIST   {r0-r6,r8,r10-r12,r15}
```

5.8.59 RN directive

The `RN` directive defines a register name for a specified register.

Syntax

The syntax of `RN` is:

`name` `RN` *`numeric-expression`*

where:

`name` is the name to be assigned to the register. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-9.

`numeric-expression`
evaluates to a register number from 0 to 15.

Usage

Use `RN` to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11      ; defines regname for register 11

sqr4        RN  r6      ; defines sqr4 for register 6
```

5.8.60 ROUT directive

The `ROUT` directive marks the boundaries of the scope of local labels. See *Local labels* on page 5-28.

Syntax

The syntax of `ROUT` is:

`{`*`name`*`} ROUT`

where:

`name` is the name to be assigned to the scope.

Usage

Use the `ROUT` directive to limit the scope of local labels. This makes it easier for your to avoid referring to a wrong label by accident. The scope of local labels is the whole AOF area if there are no `ROUT` directives in it. See *AREA directive* on page 5-38.

Use the *`name`* option to ensure that each reference is to the correct local label. If the name of a label or a reference to a label does not match the preceding `ROUT` directive, the assembler generates an error message and the assembly fails.

Example

```
            ; code
routineaA   ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code              ; this label is checked
            ; code
            BEQ     %4routineA  ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

5.8.61 SETA directive

The SETA directive sets the value of a local or global arithmetic variable.

Syntax

The syntax of SETA is:

variable-name SETA *expression*

where:

variable-name

is the name of a variable declared by a GBLA or LCLA directive.

expression

is a numeric expression. See *Numeric expressions* on page 5-89.

Usage

You must declare *variable-name* using a LCLA or GBLA directive before using SETA. Refer to *GBLA directive* on page 5-60 and *LCLA directive* on page 5-69 for more information.

Example

```
                GBLA    VersionNumber
VersionNumber   SETA    21
```

5.8.62 SETL directive

The `SETL` directive sets the value of a local or global logical variable.

Syntax

The syntax of `SETL` is:

`variable-name` `SETL` *`expression`*

where:

`variable-name`

is the name of a variable declared by a `GBLL` or `LCLL` directive.

`expression`

is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

Usage

You must declare *`variable-name`* using a `LCLL` or `GBLL` directive before using `SETL`. Refer to *GBLL directive* on page 5-61 and *LCLL directive* on page 5-70 for more information.

Example

```
        GBLL    Debug
Debug   SETL    {TRUE}
```

5.8.63 SETS directive

The `SETS` directive sets the value of a local or global string variable.

Syntax

The syntax of `SETS` is:

`variable-name` `SETS` *`string-expression`*

where:

`variable-name`

is the name of the variable declared by a `GBLS` or `LCLS` directive.

`string-expression`

is a string expression. See *String expressions* on page 5-88.

Usage

You must declare *`variable-name`* using an `LCLS` or `GBLS` directive before using `SETS`. Refer to *GBLS directive* on page 5-62 and *LCLS directive* on page 5-71 for more information.

Example

```
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

5.8.64 SUBT directive

The `SUBT` directive places a subtitle on the pages of a listing file. The subtitle is printed on each page until a new `SUBT` directive is issued.

Syntax

The syntax of `SUBT` is:

`SUBT` *`subtitle`*

where:

`subtitle` is the subtitle.

Usage

Use `SUBT` to place a subtitle at the top of the pages of a listing file. Subtitles appear in the line below the titles. See *TTL directive* on page 5-86. If you want the subtitle to appear on the first page, the `SUBT` directive must be on the first line of the source file.

Use additional `SUBT` directives to change subtitles. Each new `SUBT` directive takes effect from the top of the next page.

Example

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
        SUBT    First Subtitle  ; places a subtitle on the
                                ; second and subsequent pages
                                ; of a listing file.

        AREA    Example, CODE, READONLY
```

5.8.65 TTL directive

The `TTL` directive inserts a title at the start of each page of a listing file. The title is printed on each page until a new `TTL` directive is issued.

Syntax

The syntax of `TTL` is:

```
TTL title
```

where:

title is the title.

Usage

Use the `TTL` directive to place a title at the top of the pages of a listing file. If you want the title to appear on the first page, the `TTL` directive must be on the first line of the source file.

Use additional `TTL` directives to change the title. Each new `TTL` directive takes effect from the top of the next page.

Example

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
        AREA    Example, CODE, READONLY
```

5.8.66 WEND directive

See *WHILE directive* on page 5-87.

Syntax

The syntax of `WEND` is:

```
WEND
```

5.8.67 WHILE directive

The `WHILE` directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a `WEND` directive. See *WEND directive* on page 5-86.

Syntax

The syntax of `WHILE` is:

`WHILE` *`logical-expression`*

`code`

`WEND`

where:

`logical-expression`

is an expression that can evaluate to either `{TRUE}` or `{FALSE}`. See *Logical expressions* on page 5-90.

Usage

Use the `WHILE` directive, together with the `WEND` directive, to assemble a sequence of instructions a number of times. The number of repetitions may be zero.

You can use `IF...ENDIF` conditions within `WHILE...WEND` loops.

`WHILE...WEND` loops can be nested. See *Nesting directives* on page 5-30.

Example

```
count   SETA    1                  ; you are not restricted to
        WHILE   count <= 4         ; such simple conditions
count       SETA    count+1        ; In this case,
            ; code                 ; this code will be
            ; code                 ; repeated four times
        WEND
```

5.9 Expressions and operators

Expressions are combinations of symbols, values, unary and binary operators, and parentheses. There is a strict order of precedence in their evaluation:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

The assembler includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C. See *Unary operators* on page 5-90 and *Binary operators* on page 5-91.

5.9.1 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses. See:

- *SETS directive* on page 5-84
- *Variables* on page 5-26
- *String manipulation operators* on page 5-91
- *Unary operators* on page 5-90.

String literals consist of a series of characters contained between double quote characters. The length of a string literal is restricted by the length of the input line. See *Format of source lines* on page 5-8.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is allowed.

The value of a string expression cannot exceed 512 characters in length. It may be of zero length.

Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
                            ; sets the variable improb to the value
                            ; "literal" with the left-most four
                            ; characters of the contents of
                            ; string variable strvar2 appended
```

5.9.2 Numeric expressions

Numeric expressions consist of combinations of symbols representing numeric constants, numeric variables, actual numeric constants, binary operators, and parentheses. See *Variables* on page 5-26, *Numeric constants* on page 5-29 and *Binary operators* on page 5-91.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the program counter.

Numeric expressions evaluate to 32-bit integers. You may interpret them as unsigned numbers in the range 0 to $2^{32}–1$, or signed numbers in the range $–2^{31}$ to $2^{31}–1$. However, the assembler makes no distinction between $–n$ and $2^{32}– n$. Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

Example

```
a   SETA    256*256
    MOV     r1,#(a*22)
```

5.9.3 Register-relative and program-relative expressions

A register-relative expression evaluates to a named register plus or minus a numeric constant. It is normally a label in a register-based area combined with a numeric expression. See *^ or MAP directive* on page 5-35.

A program-relative expression evaluates to the program counter (pc) plus or minus a numeric constant. It is normally a label in a non register-based area combined with a numeric expression.

Example

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCW     value0
        ; n-1 DCW directives
        DCW     valuen          ; data+4*n points here
        ; more DCW directives
```

5.9.4 Logical expressions

Logical expressions consist of combinations of logical constants (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses. See *Boolean operators* on page 5-93.

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators. See *Relational operators* on page 5-92.

5.9.5 Unary operators

Unary operators have the highest precedence (bind most tightly) and are evaluated first. A unary operator precedes its operand and adjacent operators are evaluated from right to left.

Table 5-4 Operator precedence

Operator	Usage	Description
`?`	`?A`	Number of bytes of executable code generated by line defining symbol A.
`BASE`	`:BASE:A`	If A is a pc-relative or register-relative expression: **BASE** returns the number of its register component BASE is most useful in macros.
`INDEX`	`:INDEX:A`	If A is a register-relative expression: **INDEX** returns the offset from that base register. INDEX is most useful in macros.
`+` and `-`	`+A` `-A`	Unary plus. Unary minus. + and – can act on numeric and program-relative expressions.
`LEN`	`:LEN:A`	Length of string A.
`CHR`	`:CHR:A`	One-character string, ASCII code A.
`STR`	`:STR:A`	Hexadecimal string of A. STR returns an eight-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression.
`NOT`	`:NOT:A`	Bitwise complement of A.
`LNOT`	`:LNOT:A`	Logical complement of A.
`DEF`	`:DEF:A`	{TRUE} if A is defined, otherwise {FALSE}.

5.9.6 Binary operators

Binary operators are written between the pair of sub-expressions they operate on. Operators of equal precedence are evaluated in left to right order. The binary operators are presented below in groups of equal precedence, in decreasing precedence order.

Multiplicative operators

These are the binary operators that bind most tightly and have the highest precedence:

Table 5-5 Multiplicative operators

Operator	Usage	Explanation
`*`	`A*B`	multiply
`/`	`A/B`	divide
`MOD`	`A:MOD:B`	A modulo B

These operators act only on numeric expressions.

String manipulation operators

In the two slicing operators `LEFT` and `RIGHT`:

- `A` must be a string
- `B` must be a numeric expression.

In `CC,` `A` and `B` must both be strings.

Table 5-6 String manipulation operators

Operator	Usage	Explanation
`LEFT`	`A:LEFT:B`	the left-most B characters of A
`RIGHT`	`A:RIGHT:B`	the right-most B characters of A
`CC`	`A:CC:B`	B concatenated on to the end of A

Shift operators

The shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second.

Table 5-7 Shift operators

Operator	Usage	Explanation
ROL	A:ROL:B	rotate A left by B bits
ROR	A:ROR:B	rotate A right by B bits
SHL	A:SHL:B	shift A left by B bits
SHR	A:SHR:B	shift A right by B bits

Note

SHR is a logical shift and does not propagate the sign bit.

Addition and logical operators

The bitwise operators act on numeric expressions. The operation is performed independently on each bit of the operands to produce the result.

Table 5-8 Addition and logical operators

Operator	Usage	Explanation
AND	A:AND:B	bitwise AND of A and B
OR	A:OR:B	bitwise OR of A and B
EOR	A:EOR:B	bitwise Exclusive OR of A and B
+	A+B	add A to B
-	A-B	subtract B from A

Relational operators

Relational operators act on two operands of the same type to produce a logical value.

The operands may be:

- numeric
- program-relative

- register-relative
- strings.

Strings are sorted using ASCII ordering. String A is less than string B if it is a leading substring of string B, or if the left-most character in which the two strings differ is less in string A than in string B.

Arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

Table 5-9 Relational operators

Operator	Usage	Explanation
`=`	`A=B`	A equal to B
`>`	`A>B`	A greater than B
`>=`	`A>=B`	A greater than or equal to B
`<`	`A<B`	A less than B
`<=`	`A<=B`	A less than or equal to B
`/=`	`A/=B`	A not equal to B
`<>`	`A<>B`	A not equal to B

Boolean operators

These are the weakest binding operators with the lowest precedence.

In all three cases both A and B must be expressions that evaluate to either `{TRUE}` or `{FALSE}`.

Table 5-10 Boolean operators

Operator	Usage	Explanation
`LAND`	`A:LAND:B`	logical AND of A and B
`LOR`	`A:LOR:B`	logical OR of A and B
`LEOR`	`A:LEOR:B`	logical Exclusive OR of A and B

The Boolean operators perform the standard logical operations on their operands.

Chapter 6
Linker

This chapter describes the ARM linker, armlink. The full command syntax is given, as well as reference information on the linker itself, its input and output file formats, and memory maps. This chapter also includes reference information on scatter loading that enables you to specify the memory map of an image to the linker. This chapter contains the following information:

6.1 About the linker

The ARM linker allows you to:

- link together a collection of object files and object libraries to construct an executable image
- partially link a collection of object files into an object file that can be used as input for a link step.

6.1.1 Input to armlink

Input to the ARM linker consists of:

- one or more object files in ARM Object Format (AOF). This format is described in detail in Chapter 15 *ARM Object Format*.
- optionally, one or more object libraries as described in Chapter 14 *ARM Object Library Format*.

6.1.2 Output from armlink

Output from a successful invocation of the ARM linker is in the form of either:

- an executable image in one of several output formats: ELF, AIF, BIN. These are detailed in *Image file formats* on page 6-13.
- a consolidated object file in ARM Object Format (AOF).

Constructing an executable image

When you use the ARM linker to construct an executable image, it:

- resolves symbolic references between the input object files
- extracts object modules from object libraries to satisfy otherwise unsatisfied symbolic references
- sorts object fragments (areas) according to their attributes and names, and merges similarly attributed and named fragments into contiguous chunks
- relocates relocatable values
- generates an executable image.

Constructing a consolidated object

When you use the ARM linker to construct a consolidated object, it:

- sorts object fragments (areas) according to their attributes and names, and merges similarly attributed and named fragments into contiguous chunks
- performs pc-relative relocations between merged areas
- minimizes the symbol table
- leaves unresolved references as unresolved
- generates a consolidated object file that can be used as an input to a subsequent link step.

6.2 Command syntax

This section gives a summary of the armlink command-line options. It describes their functional groups, and then gives the complete syntax for each of these options in *armlink syntax* on page 6-6.

6.2.1 Summary of armlink options

The following lists give a brief overview of each armlink command-line option. The options are arranged into functional groups.

Accessing help and information

You can get help and information on the armlink command and its options, and the armlink version number, using the following options:

```
-help
-vsn
```

Selecting the output file and output format

You name the output file using the following option:

```
-output
```

and use one of these options to select the output file format:

```
-elf
-aof
-aif
-aif -bin
-bin
```

If no file format option is selected, the default is `-elf`.

Note

Only the `-elf` option will be supported in future releases. For this reason, use of the other file format options is deprecated. For more information on the ARM implementation of ELF, refer to the ARM ELF file format description in `c:\ARM250\PDF\specs`.

Specifying memory map information

You use the following options to specify simple memory maps:

```
-ro-base
-rw-base
```

These options define the addresses of the code and data areas.

For more complex images, you use the option:

```
-scatter
```

The concept of scatter loading is described in *About scatter loading* on page 6-21. For examples of using `-scatter`, `-ro-base`, and `-rw-base`, please refer to Chapter 10 *Writing Code for ROM* in the *ARM Software Development Toolkit User Guide*.

Note

The `-base` and `-data` options supported by previous versions of armlink are obsolete, and future versions of armlink will not support them. You should use `-ro-base` and `-rw-base` instead.

Controlling image construction

These options control debugging information, libraries, and how to include and place areas:

```
-debug / -nodebug
-nozeropad
-remove
-noremove
-dupok
-entry
-first
-last
-libpath
-scanlib / -noscanlib
```

Note

The `-workspace` option supported by previous versions of armlink is obsolete.

Generating image-related information

These options control how you extract and present information on the image:

```
-info
-map
-symbols
-xref
```

By default, the linker prints the requested information on the standard output stream, stdout. However, the information can be redirected to a file using the host stream redirection facilities or the `-list` command-line option.

Controlling the linker

These options control some aspects of how the linker operates:

```
-errors
-list
-verbose
-via
-case / -nocase
-match
-unresolved / -u
```

6.2.2 armlink syntax

The complete armlink command syntax is:

```
armlink [-help] -output file [-vsn]
[-elf | -aof | -aif | -aif -bin | -bin] [-scatter file]
[-ro-base address] [-rw-base address] [-debug | -nodebug]
[-nozeropad] [-remove | -noremove] [-dupok]
[-entry entryaddress | -entry offset+object(area)]
[-first object(area)] [-last object(area)] [-libpath path]
[-scanlib | -noscanlib ] [-info topic-list] [-map]
[-symbols file] [-xref] [-errors file] [-list file] [-verbose]
[-via file] [-case | -nocase] [-match flags]
[-unresolved symbol | -u symbol] input-file-list
```

where:

`-help` prints a summary of the command-line options.

`-vsn` displays the armlink version number.

`-output` *`file`*
specifies the name of the output file (consolidated object or executable image).

`-elf`
generates the image in ELF format. This is the default.

Future versions of the ARM linker will output images in ELF file format only. You can use the fromELF utility to convert an ELF file to another format. This utility is described in Chapter 8 *Toolkit Utilities*.

`-aof`
generates the consolidated object in AOF. Because AOF can only be used to represent an object, this option is interpreted by the linker as a request for partial linking of the input objects into a consolidated object.

`-aif`
generates the image in executable AIF format. Because `-aif` will not be supported in future releases, you are recommended to use `-elf` to produce the output file, then run the fromELF utility to convert to AIF format.

`-aif -bin`
generates the image in non-executable AIF format. Because `-aif -bin` will not be supported in future releases, you are recommended to use `-elf` to produce the output file, then run the fromELF utility to convert to AIF BIN.

`-bin`
generates the image in plain binary format. Because `-bin` will not be supported in future releases, you are recommended to use `-elf` to produce the output file, then run the fromELF utility to convert to BIN.

`-scatter` *`file`*
creates the image memory map using the scatter-loading description contained in *`file`*. The description provides grouping and placement details of the various regions, sections, and areas in the image.

`-ro-base` *`address`*
sets the execution address of the region containing the RO section of the image to *`address`*. This is also its load address. If this option is not specified, the default RO base address of 0x8000 is used, or 0x0 for binary images (unless specified by `-scatter`).

`-rw-base` *`address`*

sets the execution address of the second execution region to *`address`*. If this option is used, the image has two execution regions. If the execution address of the first execution region is not specified, the default value is used. The default address for the RW section is the end of the RO section. For example, if the RO section begins at 0x8000, and is 0x1000 bytes long, the default RW address is 0x9000.

`-debug` includes debug information in the output file. The linker adds low-level symbolic debugging data to an image (except binary images), even if the objects were compiled or assembled without any debugging information. This is the default.

`-nodebug` turns off the inclusion of debug information in the output file. The image is then smaller, but you cannot debug it at source level.

`-nozeropad` does not include zero-initialized areas in a binary image. By default, the linker includes the zero-initialized areas in the image by filling in a sequence of zeroes. However, if the creation of such areas can be postponed till run time, the linker can produce a smaller image by not including the zero-initialized areas. This option is only meaningful if the image is being output in a plain binary format (with the `-bin` option).

`-noremove` does not remove areas unreachable from the area containing the entry point.

`-remove` removes unused areas from the image. An area is considered to be used if it contains the image entry point, or if it is referred to from a used area.

You must take care not to remove interrupt handlers when using `-remove`.

`-dupok` allows duplicate symbols so that an area can be included more than once in the image. However, if `-noremove` is also specified, the image must not contain multiple copies of the area.

`-entry` *`entrypoint`*

specifies the entry point of the image. The *`entrypoint`* may be given as either:

`entry_address` an absolute address.

`offset+object(area)`

an offset within an area within a particular object. For example:

```
-entry 8+startup(C$$code)
```

There must be no spaces within the argument to `-entry`. The *`area`* and *`object`* names are matched case-insensitively. This latter form is often more convenient, and is mandatory when specifying an entry point for unused area elimination. (See the `-remove` option).

`-first object(area)`

places *`area`* from *`object`* first in the RO section of the image if it is a non ZI area. If it is a ZI area, it is placed first in the ZI section. This can be used to force an area that maps low addresses to be placed first (typically the reset and interrupt vector addresses). There must be no space between *`object`* and the following open parenthesis.

When using scatter loading, use `+FIRST` instead.

`-last object(area)`

places *`area`* from *`object`* last in the RW or RO section of the image if it is a non-ZI area. If it is a ZI area, it is placed last in the ZI section. For example, this can be used to force an area that contains a checksum to be placed last in the RW section. There must be no space between *`object`* and the following open parenthesis.

When using scatter loading, use `+LAST` instead.

`-libpath path` specifies a path that is used to search for libraries. This path overrides the path specified by the `ARMLIB` environment variable. If you do not specify a path using `-libpath`, the linker searches in the path specified by `ARMLIB`, else searches the libraries defined in the file `Lib$$Request$$library$$variant`. See *Automatic inclusion of libraries* on page 6-39 for more information on including libraries, and `$$variant`.

`-scanlib` (default) allows scanning of default libraries to resolve references.

`-noscanlib` prevents the scanning of default libraries in a link step. This is the opposite of `-scanlib`. (See also `-libpath` above).

`-info topic-list`

prints information about specified topics, where *`topic-list`* is a comma-separated list of topic keywords. A topic keyword may be one of the following:

`Totals` reports the total code and data sizes in the image. The totals are broken down into separate totals for object and library files.

`Sizes` gives a detailed breakdown of the code and data sizes for each input object and interworking veneers.

`Interwork` is ignored in this release of the linker.

`Unused` lists all unused areas, when used with the `-remove` option.

Note that spaces are not allowed between keywords in a list. For example, you can enter:

```
-info sizes,totals
```

but not:

```
-info sizes, totals
```

`-map` creates an image map listing the base and size of each constituent area.

`-symbols file` lists each symbol used in the link step (including linker-generated symbols) and its value, in the named *`file`*. A filename of minus (`-`) names the standard output stream instead of a file.

`-xref` lists cross-references between input areas.

`-errors file` redirects the diagnostics from the standard error stream to file.

`-list file` redirects the standard output stream to *`file`*. This is useful in conjunction with `-map`, `-xref`, and `-symbols`.

`-verbose` prints messages indicating progress of the link operation.

`-via file` reads a further list of input filenames and linker options from *`file`*. Each filename and option must be given on a separate line, and cannot occupy more than one line.

You can enter multiple `-via` options on the armlink command line. The `-via` options can also be included within a via file.

`-case` uses case-sensitive symbol name matching. This is the default.

`-nocase` uses case-insensitive symbol name matching.

`-match` *`flags`* sets the default and symbol-matching options so that pc-relative implies code relocation. Each option is controlled by a single bit in *`flags`*:

`0x01` matches an undefined symbol of the form `_sym` to a symbol definition of the form `sym`.

`0x02` matches an undefined symbol of the form `sym` to a symbol definition of the form `_sym`.

`0x04` matches an undefined symbol of the form `Module_Symbol` to a definition of the form `Module.Symbol`.

`0x08` matches an undefined symbol of the form `symbol_type` to a definition of the form `symbol`.

`0x10` treats all pc-relative relocation directives as relocating instructions. This is the default.

These options are usually set by configuring the linker when it is installed. Do not override options accidentally when using `-match` from the command line.

`-unresolved` *`symbol`*

matches each reference to an undefined symbol to the global definition of *`symbol`*. Note that *`symbol`* must be both defined and global, otherwise it will appear in the list of undefined symbols, and the link step will fail. This option is particularly useful during top-down development, when it may be possible to test a partially-implemented system (where the lower levels of code are missing) by connecting each reference to a missing function to a dummy function that does nothing.

This option does not display warnings.

`-u` *`symbol`* As for `-unresolved`, but this option displays warnings for each unused symbol encountered.

`input-file-list` is a space-separated list of object and library files.

6.3 Building blocks for objects and images

Figure 6-1 shows how objects and images are built.

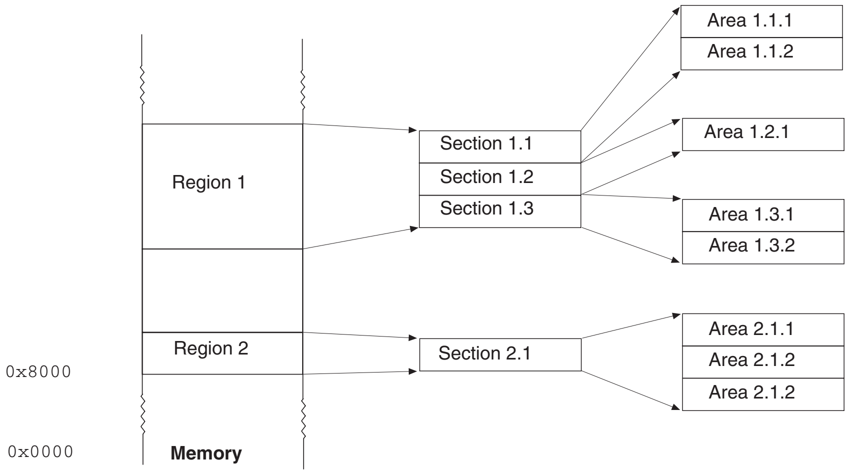


Figure 6-1 Building blocks for an image

Areas These are the code and data items from which both objects and images are built. An area contains code or initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. The attributes of an area describe whether it is read-only, read-write, executable, initialized, zero-initialized or non-initialized. Areas are further grouped into bigger building blocks called sections and regions, that are used to build images.

Sections These are sequences of areas that are contiguous in the address space and have the same attributes. A section has the same attributes as those of its constituent areas. A section containing read-only, executable areas is a read-only, executable section.

Regions These are sequences of sections contiguous in the address space. The constituent sections need not have the same attributes. A region may consist of a read-write section followed by a zero-initialized section.

In terms of these building blocks:

Object This consists of one or more areas.

Image This consists of one or more regions. The image types supported by armlink are detailed in *Image file formats* on page 6-13.

6.4 Image file formats

The ARM linker can produce an image in the following file formats:

- Executable and Linkable Format (ELF)
- ARM Image Format (AIF)
- Plain Binary format.

Note

Future versions of the ARM Linker will output images only in ELF file format. You can use the fromELF utility to convert ELF into all the other required formats. This utility is described in Chapter 8 *Toolkit Utilities*.

6.4.1 ELF format

Executable ARM ELF images conform to the TIS Portable Formats Specification version 1.1. Implementation specific details are described in the *ARM ELF File Format*, available in `ARM250\PDF\specs`. ELF is the default file format for producing an executable image. Other formats are supported to maintain backward compatibility with previous versions of the ARM linker.

6.4.2 AIF format

There are three variants of the AIF file format:

- Executable AIF format.
- Non-executable AIF format.
- Extended AIF format. This is a special type of non-executable AIF format.

See Chapter 13 *ARM Image Format* for full details of this format.

6.4.3 Plain binary format

An image in plain binary format is a sequence of bytes to be loaded into memory at a known address. The linker is not concerned with how this address is communicated to the loader, or where to enter the loaded image.

In order to produce a plain binary output, there must be:

- No unresolved symbolic references between the input objects (each reference must resolve directly or through an input library).
- An absolute base address (given by the `-ro-base` option to armlink, or in the scatter load description file). This is set to 0x0 if it is not specified.
- Complete performance of all relocation directives.

Input areas that have the read-only attribute are placed at the low-address end of the image, followed by initialized writable areas. By default, the zero-initialized areas are consolidated at the end of the file where a block of zeros of the appropriate size is written. If the `-nozeropad` linker option is specified, the application must create the zero-initialized area at runtime.

Note

If the scatter-loaded image contains zero-initialized sections, these sections must be created by the application startup code at runtime using the initialization information generated by the linker. The executable file contains just the initialization information for such sections, not the sections themselves.

6.5 Image structure

The structure of an image is defined by the number of its constituent regions and sections, and their positions in the memory map at image loading and image execution times. The structure of an image is distinct from its layout in the executable file. The executable file layout is defined by the file format.

6.5.1 Load and execution memory maps of an Image

Image regions are placed in the system memory map at load time. Before you can execute the image, you might need to move some of its regions to their execution locations, and create the zero-initialized sections at locations described by information contained in the image file.

For example, initialized read-write data may need to be copied from its load address in ROM to its execution address in RAM.

The memory map of an image has two distinct views:

- the load view
- the execution view.

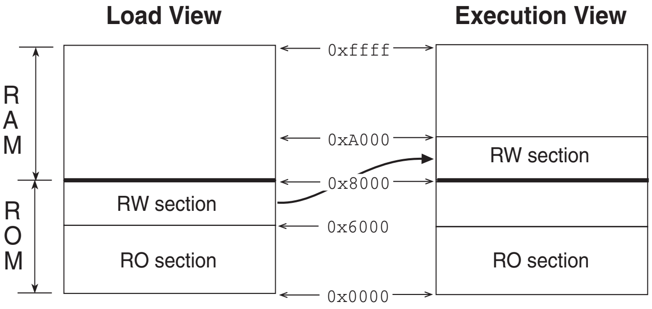


Figure 6-2 Load and execution memory maps

Table 6-1 compares the load and execution views.

Table 6-1 Comparing load and execution

Load		**Execution**	
Load view	describes each image region, section, and area in terms of the address it is located at when the image is loaded into memory, before it starts executing.	Execution view	describes each image region, section, and area in terms of the address it is located at while the image is executing.
Load address	is the address at which an area, section, or region is loaded into memory before the image containing it starts executing. The load address of an area, a section, or a region may differ from its execution address.	Execution address	is the address at which an area, section, or region is located while the image containing it is being executed.
Load section	is a section in the load address space.	Execution section	is a section in the execution address space.
Load region	is a region in the load address space.	Execution region	is a region in the execution address space.
Load image	In the load view, an image consists of one or more load regions, each of which may contain one or more load sections.	Execution image	In the execution view, an image consists of one or more execution regions, each of which may contain one or more execution sections.

6.6 Specifying an image memory map

An image can consist of any number of regions, sections, and areas. Any number of regions can have different load and execution addresses.

To construct the memory map of an image, the linker needs information on:

grouping describing how input areas are grouped into sections and regions.

placement describing where image regions should be located in the memory maps.

Depending on the complexity of the memory maps of the image, there are two ways to pass this information to the linker:

- Command-line options for simple cases.
  A simple image has at most two execution regions.
- Scatter loading for more complex cases.
  Scatter loading gives you complete control over the grouping and placement of image components by means of a scatter load description that is passed to the linker in a text file. This is described in full in *About scatter loading* on page 6-21.

6.6.1 Simple images

A simple image consists of:

- a read-only (RO) section
- a read-write (RW) section
- a zero-initialized (ZI) section.

The sections contain the input RO, RW, and ZI areas respectively. If the image does not have any RO, RW, or ZI areas, the linker does not create the corresponding RO, RW, or ZI sections.

In a simple image, you specify the execution addresses at which the Read-Only section and the Read-Write section are placed in the memory map using the following linker options:

`-RO-base` *`exec_address`*

instructs the linker to place the Read-Only section at *`exec_address`* (for example, the address of the first location in ROM)

`-RW-base` *`exec_address`*

instructs the linker to place the Read-Write section at *`exec_address`*

At application load time, the RO region is loaded at its execution address and the RW region is loaded immediately after the RO region.

The RW (data) section may contain code, because programs sometimes modify themselves (or generate code and execute it). Similarly, the RO (code) area may contain read-only data (for example string literals, floating-point constants, ANSI C **`const`** data).

Using the addresses passed to it, the linker generates the symbols required to allow the region to be copied from its load address to its execution addresses. These symbols describe the execution address and the limit of each region. They are defined independently of any input files and, along with all other external names containing `$$`, are reserved by ARM.

In a simple image, the load address of the region containing the RO section is the same as its execution address so this region never needs to be moved before execution.

Load view

In the load view, the simplest image has a single load region consisting of the RO and RW load sections placed consecutively in the memory map.

Note

The ZI section does not exist at load time (except for binary images, unless `-nozeropad` is used). It is created before execution using the description of the section contained in the image file.

Execution view

In the execution view, the image has two execution regions containing the three execution sections:

- The RW and the ZI execution sections are placed consecutively, forming one execution region.
- If no separate RW base address is given, this execution region also includes the RO execution section, and the image has only one execution region.
- If separate RO and RW base addresses are given, the RO execution section is placed separately from the RW and ZI execution sections, in its own execution region.

One execution region

In the execution view, the RO, RW, and ZI sections are laid out consecutively in a single execution region. The execution address of the region containing the RO section is the same as its load address. However, the ZI section still needs to be created before execution begins.

You specify the execution address (that is also its load address) using:

```
-ro-base execution_address
```

where `execution_address` is the execution address of the execution region.

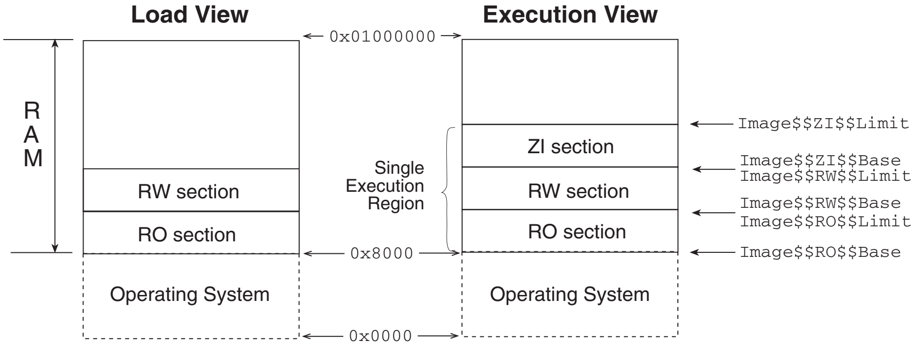


Figure 6-3 Simple image with one execution region

As an example, this approach is suitable for OS-based systems that load programs (code and data) into RAM (such as Angel on a PID board).

The symbols describing the load and/or execution addresses are listed in *Section-related symbols* on page 6-35.

Two execution regions

In Figure 6-4, there are two execution regions.

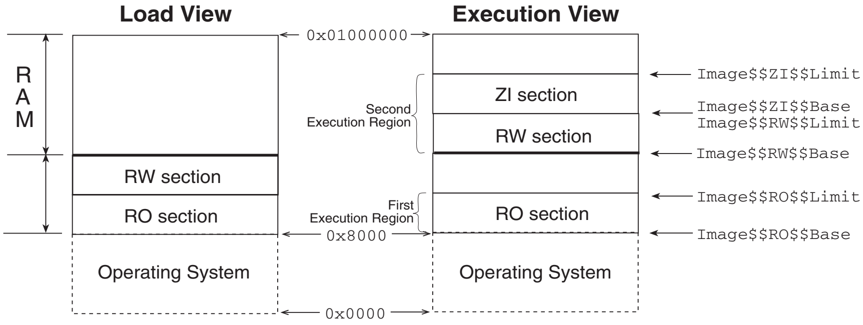


Figure 6-4 Simple image with two execution regions

Because of this, you must specify two execution addresses:

```
-ro-base execution_address_1 -rw-base execution_address_2
```

where:

execution_address_1

is the execution address of the RO execution region. The first execution region contains the RO section. The execution address of the first execution region (containing the RO section) is also the same as its load address.

execution_address_2

is the execution address of the RW execution region.
The execution address of the second execution region is different from its load address, so it needs to be moved from its load address before execution begins. The second execution region contains RW and ZI sections. The ZI section needs to be created before execution begins.

The symbols created are listed in *Section-related symbols* on page 6-35.

As an example, this approach is suitable for simple ROM-based embedded systems, or OS-based systems that load programs.

6.7 About scatter loading

The scatter loading mechanism enables you to specify the memory map of an image to the linker. Scatter loading gives you complete control over the grouping and placement of image components. It is capable of describing complex image maps consisting of multiple regions scattered in the memory map at load and execution time. Figure 6-5 on page 6-22 shows an example of a complex memory map.

An image consists of one or more regions. Each region contains one or more sections, and each section contains one or more areas. The load and execution address for a region need not be the same.

To construct the memory map of an image, the linker needs:

- grouping information describing how input areas are grouped into sections and regions
- placement information describing the addresses where image regions should be located in the memory maps.

You specify this information using a scatter load description in a text file that is passed to the linker.

For examples of using scatter loading, please refer to Chapter 10 *Writing Code for ROM* in the *ARM Software Development Toolkit User Guide*.

6.7.1 Symbols defined for scatter loading

When the linker is creating an image using a scatter load description, it defines some region-related symbols independently of any of its input files. These are described in *Region-related symbols* on page 6-34.

6.7.2 Command-line option

The linker command-line option for using scatter loading is:

```
-scatter description_file_name
```

This instructs the linker to construct the image memory map as described in `description_file_name`. The format of the description file is given in *The scatter load description file* on page 6-23.

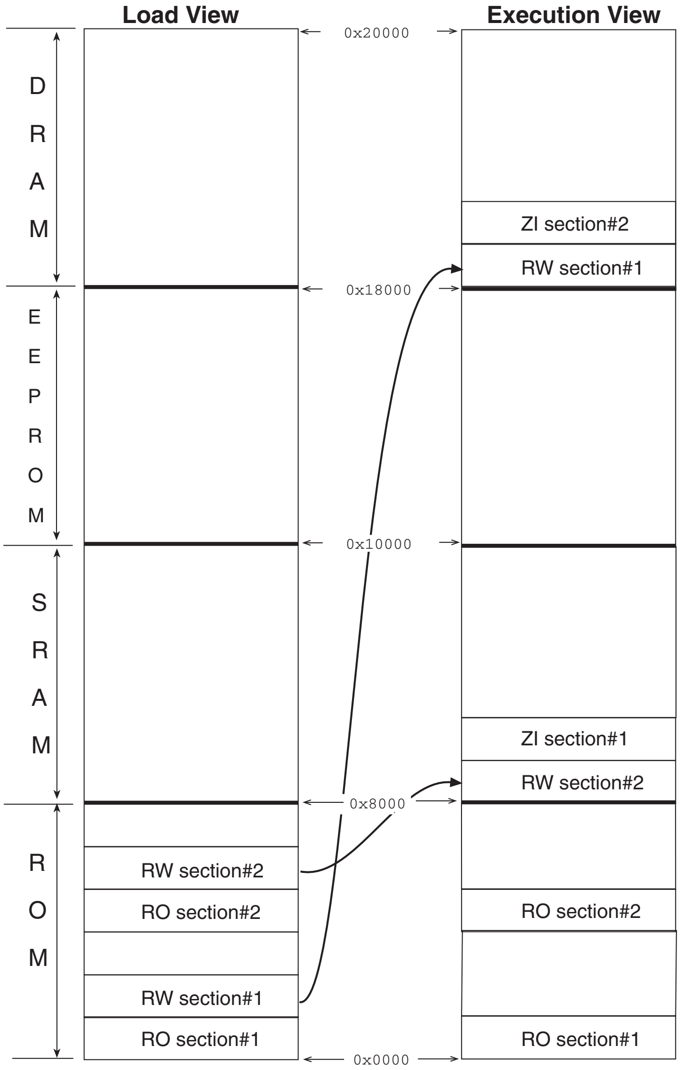


Figure 6-5 Scatter loaded memory map

6.8 The scatter load description file

An image is made up of regions and areas. Every region in the image can have a different load and execution address.

The types of region used in scatter loading are:

load region The memory occupied by a program when it has been loaded into memory, but before it starts executing can be split into a set of disjoint *load regions*. Each load region is a contiguous chunk of bytes, and contains one or more execution regions.

execution region The memory used by a program while it is executing can also be split into a set of disjoint *execution regions*. Each execution regions belongs to only one load region. An area must be in one, and only one, execution region.

6.8.1 Describing the memory map to the linker

The scatter load description is a text file that describes the memory map of the image to the linker. The description file allows you to specify:

- the number of load regions in the image
- the load address and maximum length of each load region
- the execution regions derived from each load region
- the execution address of each execution region
- the constituent areas of each execution region.

The file format reflects the hierarchy of load regions, execution regions, and areas.

Note

Sections are not used in the scatter load description file. The scatter load description file only selects areas into each region. Internally, each region is made up of sections.

The assignment of areas to regions is completely independent of the order in which patterns are written in the scatter load description.

The description itself is a sequence of tokens, whitespace, and comments, as shown in Table 6-2 on page 6-24.

Table 6-2 Scatter load description

Item	Description	
Special character	Single-characters with special significance are:	
	`(`	`LPAREN`
	`)`	`RPAREN`
	`{`	`LBRACE`
	`}`	`RBRACE`
	`"`	`QUOTE`
	`,`	`COMMA`
	`+`	`PLUS`
	`;`	`SEMIC`

Tokens	Tokens are:	`LPAREN`
		`RPAREN`
		`LBRACE`
		`RBRACE`
		`QUOTE`
		`COMMA`
		`PLUS`
		`SEMIC`

Comments	Comments begin with a `SEMIC` and extend to the end of the current line. This means that a `WORD` cannot begin with a `SEMIC` (unless the `WORD` and `SEMIC` are enclosed in `QUOTE`s).	
Numbers	A `NUMBER` encodes a 32-bit unsigned value, and has one of the forms:	
	Prefix:	Number:
	O	octal-digit+
	&	hex-digit+
	ox	hex-digit+
	Ox	hex-digit+
		decimal-digit+
Word	A `WORD` is an alternation of quoted and unquoted `WORD`-segments:	
	Unquoted `WORD` segment	terminates on the first character in the set {Whitespace, `LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, `COMMA`, `PLUS`, `QUOTE`}
	Quoted `WORD` segment	is enclosed by `QUOTE` characters and may contain any characters except *`newline`*. All other characters of which `isspace()` is true are translated to `space`. Two consecutive `QUOTE`s stand for the literal `QUOTE` character and do not begin or end a quoted `WORD`-segment.

6.8.2 Structure of the description file

Structurally, a scatter load description is a sequence of load region descriptions:

```
Scatter-description ::= load-region-description+
```

Load region description

A load region has:

- a name
- a base address
- an optional maximum size
- a non-empty list of execution regions.

The linker allows an empty description.

The syntax is:

```
load-region-description ::=
      load-region-name base-address [ max-size ]
            LBRACE execution-region-description+ RBRACE
```

where:

`load-region-name`

names the load region. Only the first 31 characters are significant. Use fewer characters if your system has a shorter limit on directory entries. The linker uses the first 31 characters to produce base and limit symbols for the region.

In multi-file output formats (for example `-bin`), *`load-region-name`* is used to name the file containing initializing data for this load region.

`base-address` is the address where the contents of the region are loaded. It must be a word-aligned `NUMBER`, so &1234ABDC, 0x1e4, 4000, and 0 are acceptable, but 1234CD is not.

`max-size` is an optional `NUMBER`. If `max-size` is specified, the linker generates an error if the region has more than `max-size` bytes allocated to it.

`execution-region-description`

is described in the following subsection.

Execution region description

An execution region is described by a name and a base address.

The syntax is:

```
execution-region-description ::=
        exec-region-name base-designator
                LBRACE area-description* RBRACE

base-designator ::= base-address  | + offset
```

where:

exec-region-name

names the execution region. Only the first 31 characters are significant. Use fewer characters if your system has a shorter limit on directory entries.

base-designator describes the base address:

base-address is the address at which objects in the region should be linked. It must be a word-aligned NUMBER.

+ *offset* describes a base address that is *offset* bytes beyond the end of the preceding execution region. The length of a region is always a multiple of four bytes, so *offset* must also be a multiple of four bytes. If there is no preceding execution region (that is, if this is the first in the load region) then + *offset* means *offset* bytes after the base of the containing load region.

`area-description`

is described in the following subsection.

Area description

An `area-description` is a pattern that identifies areas by:

- module name (object file name, library member name, or library file name)
- area name, or area attributes such as `READ-ONLY`, or `CODE`.

Note

Only areas that match both the `module-selector-pattern` and at least one *`area-selector`* are included in the execution region.

If you omit `LPAREN` *`area-selectors`* `RPAREN`, the default is `+RO`.

The syntax is:

```
area-description ::=
      module-selector-pattern [ LPAREN area-selectors RPAREN ]
```

where:

`module-selector-pattern`

is a pattern constructed from literal text, and the wildcard characters `*` (matches 0 or more characters) and `?` (matches any single character). For example:

```
*armlib.*
```

An area matches a *`module-selector-pattern`* if:

- The name of the object file containing the area or name of the library member (with no leading pathname) matches the *`module-selector-pattern`*.
- The full name of the library from which the area was extracted matches the *`module-selector-pattern`*.

  Matching is case-insensitive, even on hosts with case-sensitive file naming.

`area-selectors`

is a comma-separated list of expressions. Each expression is a pattern against which the area name, or the name of an attribute you want the selected area to have, is matched. In the latter case the name must be preceded by a plus character (+). You may omit any comma immediately followed by a `PLUS`.

See *ARM/Thumb interworking veneers* on page 6-29 for information on assigning ARM/Thumb interworking veneers to an execution region.

```
area-selectors ::=
      (PLUS area-attrs | area-pattern )([ COMMA ]
      PLUS area-attrs| COMMA area-pattern)*
```

where:

`area-pattern` is a pattern that is matched case-insensitively against the area name. It is constructed from literal text, and the wildcard characters `*` (matches 0 or more characters) and `?` (matches any single character).

`area-attrs` is an attribute selector matched against the area attributes. Each *`area-attrs`* follows a `PLUS` keyword or `+` character.

All selectors are case insensitive. The following selectors are recognized:

- `RO-CODE`
- `RO-BASED-DATA`
- `RO` (includes `RO-CODE` and `RO-DATA`)
- `RO-DATA` (includes `RO-BASED-DATA`)
- `RW-STUB-DATA` (shared library stub data)
- `RW-DATA` (includes `RW-BASED-DATA` and `RW-STUB-DATA`)
- `RW` (includes `RW-CODE` and `RW-DATA`)
- `ZI`
- `ENTRY` (the area containing the `ENTRY` point)

`RO-NOTBASED-DATA` cannot be specified directly. `RO-BASED-DATA` must be selected in one region and (less specifically) `RO-DATA` in another.

The following synonyms are recognized:

- `CODE (= RO-CODE)`
- `CONST (= RO-DATA)`
- `TEXT (= RO)`
- `DATA (= RW)`
- `BSS (= ZI)`

The following pseudo-attributes are recognized:

- `FIRST`
- `LAST`

`FIRST` and `LAST` can be used to mark the first and last areas in an execution region if the placement order is important (for example, if the `ENTRY` must be first and a checksum last). The first occurrence of `FIRST` or `LAST` as an *`area-attrs`* terminates the list.

ARM/Thumb interworking veneers

ARM/Thumb interworking veneers are built in an area called `IWV$$Code`. You can assign this area to an execution region just like any other area using the *`area-selector`*:

```
*(IWV$$Code)
```

Although there is no associated *`module-selector-pattern`*, `*` still matches as there is only one `IWV$$Code` area, so this selection is unambiguous.

6.8.3 Resolving multiple matches

If an area matches more than one execution region, the matches are resolved as described below. If a unique match cannot be found, armlink faults the scatter description. Each area is selected by a *`module-selector-pattern`* and an *`area-selector`*.

In the following explanation:

- `m1` and `m2` represent module-selector-patterns.
- `s1` and `s2` represent area-selectors.
- the term ‘iff’ means *if and only if*
- the greater than operator (>) denotes *more specific*. For example:

  `m1,s1 > m2,s2`

  means that the `m1,s1` pair is more specific than the `m2,s2` pair.

In the case of multiple matches, the linker determines which region to assign the area to on the basis of which *`module-selector-pattern`*, *`area-selector`* pair is the most specific.

For example, if area A matches `m1,s1` for execution region R1, and `m2,s2` for execution region R2, the linker:

- assigns A to R1 iff `m1,s1 > m2,s2`
- assigns A to R2 iff `m2,s2 > m1,s1`
- diagnoses the scatter description as faulty if neither `m1,s1 > m2,s2` nor `m2,s2 > m1,s1`.

The linker determines the most specific *module-selector-pattern,area-selector* pair in the following way:

- For the module selector patterns:

  ```
  m1 > m2 iff (text(m1) matches pattern(m2)) &&
              !(text(m2) matches pattern(m1))
  ```

- For the area selectors:
  - If `s1` and `s2` are both patterns matching area names, the same definition as for module selector patterns is used.
  - If one of `s1`, `s2` matches the area name and the other matches the area attributes then `s1` and `s2` are unordered and the description is diagnosed as faulty.
  - If both `s1` and `s2` match area attributes, `s1 > s2` is defined by:

    ```
    ENTRY > RO-CODE > RO
    ENTRY > RO-BASED-DATA > RO-DATA > RO
    ENTRY > RW-CODE > RW
    ENTRY > RW-BASED-DATA > RW-DATA > RW
    ENTRY > RW-STUB-DATA > RW-DATA > RW
    ```

    No other members of the `s1 > s2` relation between area attributes exist.

- For the *module-selector-pattern,area-selector* pair:

  `m1,s1 > m2,s2` if and only if either of the following is true:
  - `s1` is a literal area name (that is, it contains no pattern characters) and `s2` matches area attributes other than `+ENTRY`
  - `(m1 > m2) || !(m2 > m1) && (s1 > s2).`

The consequences of this matching strategy include:

- Descriptions are independent of the order in which they are written.

- Generally, the more specific the description of an object is, the more specific the description of the areas it contains. *area-selectors* are not examined unless:
  - Object selection is inconclusive.
  - One selector fully names an area and the other selects by attribute.

    In this case, the explicit area name is more specific than any attribute other than `ENTRY` (that selects exactly one area from one object). This is the case even if the object selector associated with the area name is less specific than that associated with the attribute.

6.8.4 Obsolete features

The following features are obsolete:

- `ROOT`
- `ROOT-DATA`
- `OVERLAY`.

Note

These are supported by the current version of armlink for backwards compatibility only and will not be supported in future versions of armlink. Using these features in a description file generates a warning from armlink.

6.9 Area placement and sorting rules

The linker sorts all the areas within a region according to their attributes. Areas with identical attributes form a contiguous block within the region. Each contiguous block of areas is defined as a section.

You need not specify explicitly the addresses or number of sections within a region, because this is done by the linker. There are as many sections in a region as there are non-empty area attribute sets. The base address of each section is governed by the sorting order defined by the linker.

While generating an image, the ARM linker sorts the input areas in the following order:

- by attribute
- by area name
- by their positions in the input list, except where overridden by a `-first` or `-last` option. This is described in *Using FIRST and LAST to place areas* on page 6-33.

By default, the ARM linker creates an image consisting of an RO, an RW, and optionally a ZI, section. The RO section can be protected at runtime on systems that have memory management hardware.

Page alignment of the RO and RW sections of the image can be forced using the area alignment attribute of AOF areas. You set this using the `ALIGN` attribute of the ARM assembler `AREA` directive (see *AREA directive* on page 5-38).

6.9.1 Ordering areas by attribute

Portions of the image associated with a particular language runtime system are collected together into a minimum number of contiguous regions. (This applies particularly to code regions that may have associated exception handling mechanisms.) More precisely, the linker orders areas by attribute as follows:

- read-only code
- read-only based data
- read-only data
- read-write code
- based data
- other initialized data
- zero-initialized (uninitialized) data
- debugging tables (optional).

Areas that have the same attributes are ordered by their names. Names are considered to be case-sensitive, and are compared in alphabetical order, using the ASCII collation sequence for characters.

Identically attributed and named areas are ordered according to their relative positions in the input list.

As a consequence of these rules, the positioning of identically attributed and named areas included from libraries is not predictable. However, if library L1 precedes library L2 in the input list, all areas included from L1 will precede each area included from L2. If more precise positioning is required, you can extract modules manually, and include them in the input list.

6.9.2 Using FIRST and LAST to place areas

Within a region, all read-only code areas are contiguous and form a section that must precede the section containing all the read-only based data areas.

If you are not using scatter loading, you use the `-first` and `-last` linker options to place areas.

If you are using scatter loading, you use the pseudo-attributes `FIRST` and `LAST` in the scatter load description file to mark the first and last areas in an execution region if the placement order is important (for example, if the `ENTRY` must be first and a checksum last).

However, `FIRST` and `LAST` must not violate the basic attribute sorting order. This means that an area can be first (or last) in the execution region if the execution section in which it is contained is the first (or last) section in the region. For example, in an execution region containing any read-only areas, the `FIRST` area must be a read-only area. Similarly, if the region contains any ZI data, the `LAST` area must be a ZI area.

Within each section, areas are sorted according to their names, and then by their positions in the input order.

6.9.3 Aligning areas

Once areas have been ordered and the base address has been fixed, the linker may insert padding to force each area to start at an address that is a multiple of:

$2^{\texttt{(area alignment)}}$

Areas are commonly aligned at word boundaries.

6.10 Linker-defined symbols

The ARM Linker defines some symbols independently of any of its input files. These symbols contain the character sequence `$$` and are reserved by ARM along with all other external names containing this sequence.

These symbols are used to specify region-base-addresses, section-base-addresses, and area-base-addresses and their limits. They can be imported and used as relocatable addresses by your assembly language programs, or referred to as **`extern`** symbols from your C or C++ source code. Examples of such usage are provided in Chapter 10 *Writing Code for ROM* in the *ARM Software Development Toolkit User Guide*.

6.10.1 Region-related symbols

These symbols are generated when the linker is creating an image using a scatter loading description. The description names all the load and execution regions in the image, besides providing their load and execution addresses. The scatter load description file is explained in *The scatter load description file* on page 6-23.

The linker generates three symbols for every region present in the image, shown in Table 6-3.

Table 6-3 Region-related linker symbols

Symbol	Description
`Load$$`*`region_name`*`$$Base`	the load address of the region
`Image$$`*`region_name`*`$$Base`	the execution address of the region
`Image$$`*`region_name`*`$$Length`	the execution region length in bytes (multiple of 4)

For every region containing a ZI section, the linker generates two additional symbols, shown in Table 6-4.

Table 6-4 Additional symbols for ZI sections

Symbol	Description
`Image$$`*`region_name`*`$$ZI$$Base`	the execution address of the ZI section in this region.
`Image$$`*`region_name`*`$$ZI$$Length`	the length of the ZI section in bytes (multiple of 4)

Note

A scatter load image is not padded with zeros, and requires the ZI sections to be created dynamically. This means that there is no need for a load address symbol for ZI data.

6.10.2 Section-related symbols

The symbols shown in Table 6-5 are generated when the linker is creating a simple image that has three sections (RO, RW and ZI) that are combined into one or two execution regions, when `-ro-base` and/or `-rw-base` are specified.

Table 6-5 Section-related linker symbols

Symbol	Description
`Image$$RO$$Base`	Address of the start of the RO section: • If the image has one execution region, `Image$$RW$$Base` is the same as `Image$$RO$$Limit.` • If the image has two execution regions, `Image$$RW$$Base` is different from `Image$$RO$$Limit.`
`Image$$RO$$Limit`	Address of the byte beyond the end of the RO section
`Image$$RW$$Base`	Address of the start of the RW section
`Image$$RW$$Limit`	Address of the byte beyond the end of the ZI section.
`Image$$ZI$$Base`	Address of the start of the ZI section
`Image$$ZI$$Limit`	Address of the byte beyond the end of the ZI section.

Note

Section-related symbols do not contain useful information if scatter loading is used to specify region information. Code that uses section-related symbols will not produce expected results. In such cases, only the region-related symbols described in *Region-related symbols* on page 6-34 should be used.

6.10.3 Area-related symbols

For every area present in the image, the linker generates the symbols in Table 6-6.

Table 6-6 Area-related linker symbols

Symbol	Description
`Areaname$$Base`	Address of the start of the consolidated area called `Areaname`.
`Areaname$$Limit`	Address of the byte beyond the end of the consolidated area called `Areaname`.

6.11 Including library members

An object file may contain references to external objects (functions and variables). The linker attempts to resolve these references by matching them to definitions found in other object files and libraries. A library is a collection of AOF files stored in an ARM Library Format file. Usually, at least one library file is specified in the input list. The important differences between object files and libraries are:

- each object file in the input list appears in the output unconditionally, whether or not anything refers to it
- a member from a library is included in the output if, and only if, an object file or an already-included library member makes a non-weak reference to it.

6.11.1 Processing the input file list

The linker processes its input list as follows:

1. The object files are linked together, ignoring the libraries. Usually there will be a resultant set of references to as yet undefined symbols. Some of these may be weak. These are allowed to remain unsatisfied, and do not cause library members to be loaded.

2. The libraries are then processed in the order that they appear in the input file list:

   a. The library is searched for members containing symbol definitions that match currently unsatisfied, non-weak references.

   b. Each such member is loaded, satisfying some unsatisfied references, possibly including weak ones. This may create new unsatisfied references, again, possibly including weak ones.

   c. The search of the library is repeated until no further members are loaded from the library.

   d. The libraries are re-processed as described in Step 2 until no further members are loaded.

If any non-weak reference remains unsatisfied at the end of a linking operation, other than one that generates partially-linked, relocatable AOF, the linker generates an error message.

While processing the object files and libraries specified to the linker on its command line, a warning is given if symbols are seen requesting different variants of the same library, but all requested variants are added to the list of requested libraries (in the order they are requested by the input files).

The libraries in the list are searched only if there are still unsatisfied non-weak references after all specified objects and libraries have been loaded. They are obtained from the directory specified to the linker by use of a `-libpath` argument (or configured as its library path using the graphical configuration tool), or failing that, from the directory which is the value of the environment variable `ARMLIB`.

6.11.2 Including library members

To forcibly include a library member, put the name(s) of the library member(s) in parentheses after the library name. There must be no space between the library name and the opening parenthesis. Multiple member names must be separated by commas. There must be no space in the list of member names.

6.12 Automatic inclusion of libraries

The ARM linker automatically searches for a C library that matches the attributes of the object files being linked. To do this, the filenames of the C libraries are annotated with letters and digits to identify them. The annotation has the form:

`armlib_<apcs_variant>.<bits><bytesex>`

The compiler and assembler generate a weak reference to symbols with names `Lib$$Request$$library$$`*`variant`* for required libraries, where *`variant`* is determined by the APCS options in use. this is described above, except that armcc with `apcs/interwork` generates a reference to the Thumb interworking libraries. The ARM and Thumb C compilers and assemblers require only the armlib library.

6.12.1 For ARM libraries

`apcs_variant` is the concatenation of a hardware floating-point option:

`h` hardware floating-point, instruction set 3.

`r` hardware floating-point, instruction set 3, fp arguments in fp registers.

`2` hardware floating-point, instruction set 2.

`z` hardware floating-point, instruction set 2, fp arguments in fp registers.

with a software stack checking option:

`c` no software stack checking.

with an interworking option:

`i` compiled for ARM/Thumb interworking.

and a frame pointer option:

`n` does not use a frame pointer.

`bits` is 32.

`bytesex` is either:

`l` little-endian

`b` big-endian

Example

The following example is an ARM little-endian C library, with no software stack checking and no frame pointer:

```
armlib_cn.32l
```

6.12.2 For Thumb libraries

`apcs_variant` is the concatenation of a software stack checking option:

`s`	software stack checking.

and an interworking option:

`i`	compiled for ARM/Thumb interworking.

`bits` is 16

`bytesex` is either:

`l`	little-endian
`b`	big-endian

Example

The following example is a Thumb little-endian C library compiled for interworking:

```
armlib_i.16l
```

Note

Not all combinations are possible. Only a subset of the possible combinations are made by ARM as part of a release.

6.13 Handling relocation directives

This section describes how the linker implements the relocation directives defined by ARM Object Format.

6.13.1 The subject field

A relocation directive describes the relocation of a single subject field, that may be:

- a byte
- a halfword (2 bytes, halfword-aligned)
- a word (4 bytes, word-aligned)
- a value derived from a suitable sequence of instructions.

The relocation of a word value cannot overflow. In the other cases, overflow is diagnosed by the linker. This is described in *The relocation of instruction sequences* on page 6-42.

6.13.2 The relocation value

A relocation directive refers either to:

- the value of a symbol
- the base address of an AOF area in the same object file as the AOF area containing the directive.

This value is called the relocation value, and the subject field is modified by it, as described in the following subsections.

6.13.3 PC-relative relocation

A pc-relative relocation directive requests the following modification of the subject field:

```
subject-field = subject_field + relocation_value
                - base_of_area_containing(subject_field)
```

A special case of pc-relative relocation occurs when the relocation value is specified to be the base of the area containing the subject field. In this case, the relocation value is not added. This caters for a pc-relative branch to a fixed location, for example:

```
subject-field =subject_field-
               base_of_area_containing(subject_field)
```

6.13.4 Additive relocation

A plain additive relocation directive modifies the subject field as follows:

```
subject_field = subject_field + relocation_value
```

6.13.5 Based area relocation

A based area relocation directive relocates a subject field by the offset of the relocation value within the consolidated area containing it:

```
subject_field = subject_field + relocation_value
                    - base_of_area_group_containing(relocation_value)
```

6.13.6 The relocation of instruction sequences

The linker recognizes that the following instruction sequences define a relocatable value:

- an ARM `B` or `BL` instruction
- a Thumb `BL` instruction pair.

If bit 0 of the relocation offset is set, the linker relocates a Thumb instruction sequence. The only Thumb instruction sequence that can be relocated is a `BL` instruction pair. In the relocation of a `B` or `BL` instruction, word offsets are converted to and from byte offsets.

Chapter 7
ARM Symbolic Debugger

The ARM Symbolic Debugger, armsd, is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. This chapter contains the following information:

7.1 About armsd

The ARM symbolic debugger can be used to debug programs assembled or compiled using the ARM assembler, and the ARM C compiler, if those programs have been produced with debugging enabled. A limited amount of debugging information can be produced at link time, even if the object code being linked was not compiled with debugging enabled. The symbolic debugger is normally used to run ARM Image Format images.

7.1.1 Selecting a debugger

Armsd supports:

- remote debugging
- debugging using the ARMulator
- remote debugging using ADP.

7.1.2 Automatic command execution on startup

The symbolic debugger obeys commands from an initialization file, if one exists, before it reads commands from the standard input. The initialization file is called `armsd.ini`.

The current directory is searched first, and then the directory specified by the environment variable `HOME`.

7.2 Command syntax

You invoke armsd using the command given below. Underlining is used to show the permitted abbreviations.

The full list of commands available when armsd is running is given in *Alphabetical list of armsd commands* on page 7-10.

7.2.1 Command-line options

```
armsd [-help] [-vsn] [-little | -big] [-proc name] [-fpe | -nofpe]
[-symbols] [-o name] [-script name] [-exec] [-iname] [-rem |
-armul | -adp options] image_name arguments
```

where:

`-help` gives a summary of the armsd command-line options.

`-vsn` displays information on the armsd version.

`-little` specifies that memory should be little-endian.

`-big` specifies that memory should be big-endian.

`-proc` *`name`* specifies the cpu type.

`-fpe` instructs the ARMulator to load the FPE on startup.

`-nofpe` instructs the ARMulator not to load the FPE on startup. When testing code compiled using the floating-point library, you may not want to load the FPE.

`-symbols` loads an image file containing debug information but does not download the image.

`-o` *`name`* writes output from the debuggee to the named file.

`-script` *`name`* takes commands from the named file (reverts to stdin on reaching EOF).

`-exec` asks the debugger to execute immediately and quit when execution stops.

`-i`*`name`* adds *`name`* to the set of paths to be searched to find source files.

`-rem` selects remote debugging. By default this will be ADP.

`-armul` selects the software ARM Emulator (ARMulator).

`-adp` *`options`*
selects remote debugging using ADP. Select one of the following options to specify the ADP port to use:

`-port` *`expr`*
selects serial communications, where *`expr`* can be any of:

`1`
`2`
`device_names``=1`
`s=2`
`s=`*`device_name`*

To select serial and parallel communications, *`expr`* can be:

`s=`*`n`*`,p=`*`m`*

where *`n`* and *`m`* can be 1, 2 or a device name.

To select ethernet communications, *`expr`* is:

`e=`*`id`*

where *`id`* is the ethernet address of the target board.

In the case of serial and/or parallel communications, `h=0` may be prefixed to the port expression. This switches off the heartbeat feature of ADP.

`-linespeed` *`n`*
sets the line speed to *`n`*.

`-loadconfig` *`name`*
specifies the file containing configuration data to be loaded.

`-selectconfig` *`name version`*
specifies the target configuration to be used.

`-reset`
resets the target processor immediately (if supported for target).

`-clock` *`n`*
specifies the clock speed in Hz (suffixed with K or M) for the ARMulator. This is only valid with an `armsd.map` file.

`image_name`
gives the name of the file to debug. You can also specify this information using the load command. See *load* on page 7-32 for more information.

`arguments` give program arguments. You can also specify this information using the `load` command. See *load* on page 7-32 for more information.

7.3 Running armsd

This section lists all armsd commands. They are first briefly listed in functional groups, and then are explained fully in *Alphabetical list of armsd commands* on page 7-10.

The functional groups are:

- Symbols
- Controlling execution
- Program context
- Low-level debugging
- Coprocessor support
- Profiling commands
- Miscellaneous.

The semicolon character (;) separates two commands on a single line.

Note

The debugger queues commands in the order it receives them, so that any commands attached to a breakpoint are not executed until all previously queued commands have been executed.

7.3.1 Symbols

These commands allow you to view information on armsd symbols:

`symbols` lists all symbols (variables) defined in the given or current context, along with their type information.

`variable` provides type and context information on the specified variable (or structure field).

`arguments` shows the arguments that were passed to the current procedure, or another active procedure.

7.3.2 Controlling execution

These commands allow you to control execution of programs by setting and clearing watchpoints and breakpoints, and by stepping through instructions and statements:

`break` adds breakpoints.

`call` calls a procedure.

`go` starts execution of a program.

`istep` steps through one or more instructions.

`load` loads an image for debugging.

`reload` reloads the object file specified on the armsd command line, or with the last `load` command.

`return` returns to the caller of the current procedure (passing back a result).

`step` steps execution through one or more statements.

`unbreak` removes a breakpoint.

`unwatch` clears a watchpoint.

`watch` adds a watchpoint.

7.3.3 Reading and writing memory

These commands allow you to set and examine program context:

`getfile` reads from a file and writes the content to memory.

`putfile` writes the contents of an area of memory to a file.

7.3.4 Program context

These commands allow you to set and examine program context:

`where` prints the current context as a procedure name, line number in the file, filename and the line of code.

`backtrace` prints information about all currently active procedures.

`context` sets the context in which the variable lookup occurs.

`out` sets the context to be the same as that of the current context's caller.

`in` sets the context to that called from the current level.

7.3.5 Low-level debugging

These commands allow you to select low-level debugging and to examine and display the contents of memory, registers, and low-level symbols:

`language` sets up low-level debugging if you are already using high-level debugging.

`registers` displays the contents of ARM registers 0 to 14, the program counter (pc) and the status flags contained in the processor status register (psr).

`fpregisters`

displays the contents of the eight floating-point registers f0 to f7 and the floating-point processor status register FPSR.

`examine` allows you to examine the contents of the memory between a pair of addresses, displaying it in both hexadecimal and ASCII formats, with 16 bytes per line.

`list` displays the contents of the memory between a specified pair of addresses in hexadecimal, ASCII and instruction format, with four bytes (one instruction) per line.

`find` finds all occurrences in memory of a given integer value or character string.

`lsym` displays low-level symbols and their values.

7.3.6 Coprocessor support

The symbolic debugger's coprocessor support enables access to registers of a coprocessor through a debug monitor that is ignorant of the coprocessor. This is only possible if the registers of the coprocessor are read (if readable) and written (if writable) by a single coprocessor data transfer (CPDT) or a coprocessor register transfer (CPRT) instruction in a non-user mode. For coprocessors with more unusual registers, there must be support code in a debug monitor.

`coproc` describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display.

`cregdef` describes how the contents of a coprocessor register are formatted for display.

`cregisters` displays the contents of all readable registers of a coprocessor, in the format specified by an earlier `coproc` command.

`cwrite` writes to a coprocessor register.

7.3.7 Profiling commands

The following commands allow you to start, stop, and reset the profiler, and to write profiling data to a file.

`pause` prompts you to press a key to continue.

`profclear` resets profiling counts.

`profon` starts collecting profiling data.

`profoff` stops collecting profiling data.

`profwrite` writes profiling information to a file.

7.3.8 Miscellaneous commands

These are general commands.

`!` passes the following command to the host operating system.

`|` introduces a comment line.

`alias` defines, undefines, or lists aliases. It allows you to define your own symbolic debugger commands.

`comment` writes a message to `stderr`.

`help` displays a list of available commands, or help on a particular command.

`log` sends the output of subsequent commands to a file as well as the screen.

`obey` executes a set of debugger commands which have previously been stored in a file, as if they were being typed at the keyboard.

`print` examines the contents of the debugged program’s variables.

`type` types the contents of a source file, or any text file, between a specified pair of line numbers.

`while` is part of a multi-statement line.

`quit` terminates the current symbolic debugger session and closes any open log or obey files.

7.4 Alphabetical list of armsd commands

This section explains how the armsd command syntax is annotated, and lists the terminology used. Every armsd command is listed and explained, starting with the *! command* on page 7-12.

7.4.1 Annotating the command syntax

`typewriter` Shows command elements that you should type at the keyboard.

<u>type</u>writer Underlined text shows the permitted abbreviation of a command.

`typewriter` Represents an item such as a filename or variable name. You should replace this with the name of your file, variable, and so on.

`{}` Items in braces are optional. The braces are used for clarity and should not be typed.

`*` A star (*) following a set of braces means that the items in those braces can be repeated as many times as required. Many command names can be abbreviated. The braces here show what can be left out. In the one case where braces are required by the debugger, these are enclosed in quote marks in the syntax pattern.

7.4.2 Names used in syntax descriptions

These terms are used in the following sections for the command syntax descriptions:

context The activation state of the program. See *Variable names and context* on page 7-49.

expression An arbitrary expression using the constants, variables and operators described in *Expressions* on page 7-51. It is either a low-level or a high-level expression, depending on the command.

Low-level Low-level expressions are arbitrary expressions using constants, low-level symbols and operators. High-level variables may be included in low-level expressions if their specification starts with `#` or `$`, or if they are preceded by `^`.

High-level High level expressions are arbitrary expressions using constants, variables and operators. Low-level symbols may be included in high-level expressions by preceding them with `@`.

The `list`, `find`, `examine`, `putfile`, and `getfile` commands require low-level expressions as arguments. All other commands require high-level expressions.

location A location within the program (see *Program locations* on page 7-50).

variable A reference to one of the program's variables. Use the simple variable name to look at a variable in the current context, or add more information as described in *Variable names and context* on page 7-49 to see the variable elsewhere in the program.

format is one of:

- `hex`
- `ascii`
- `string`

  This is a sequence of characters enclosed in double quotes ("). A backslash (\) may be used as an escape character within a string.
- A C `printf()` function format descriptor. Table 7-1 shows some common descriptors.

Table 7-1 Format descriptors

Type	Format	Description

int	
	`%d`
	`%u`
	`%x`
		Only use this if the expression being printed yields an integer:
		Signed decimal integer (default for integers)
		Unsigned integer
		Hexadecimal (lowercase letters)
		same as hex

char	`%c`	Character (same as ascii) Only use this if the expression being printed yields an integer.
char *	`%s`	Pointer to character (same as string) Only use this for expressions which yield a pointer to a zero-terminated string.
void *	`%p`	Pointer (same as %.8x), for example, 00018abc This can be used with any kind of pointer.

float	
	`%e`
	`%f`
	`%g`
		Only use this for floating-point results:
		Exponent notation, for example, 9.999999e+00
		Fixed point notation, for example, 9.999999
		General floating-point notation, for example, 1.1, 1.2e+06

7.4.3 ! command

The `!` command gives access to the command line of the host system without quitting the debugger.

Syntax

The syntax of `!` is:

`!`*`command`*

where:

`command` is the operating system command to execute.

Usage

Any command whose first character is ! is passed to the host operating system for execution.

7.4.4 | command

The `|` command introduces a comment line.

Syntax

The syntax of `|` is:

`|`*`comment`*

where:

`comment` is a text string

Usage

This command allows you to annotate your armsd file.

7.4.5 alias

The `alias` command defines, undefines, or lists aliases. It allows you to define symbolic debugger commands.

Syntax

The syntax of `alias` is:

```
alias {name{expansion}}
```

where:

name is the name of the alias.

expansion is the expansion for the alias.

Usage

If no arguments are given, all currently defined aliases are displayed. If expansion is not specified, the alias named is deleted. Otherwise expansion is assigned to the alias name.

The expansion may be enclosed in double quotes (") to allow the inclusion of characters not normally permitted or with special meanings, such as the alias expansion character (‘) and the statement separator (;).

Aliases are expanded whenever a command line or the command list in a `do` clause is about to be executed.

Words consisting of alphanumeric characters enclosed in backquotes (‘) are expanded. If no corresponding alias is found they are replaced by null strings. If the character following the closing backquote is non-alphanumeric, the closing backquote may be omitted. If the word is the first word of a command, the opening backquote may be omitted. To use a backquote in a command, precede it with another backquote.

7.4.6 arguments

The `arguments` command shows the arguments that were passed to the current, or other active procedure.

Syntax

The syntax of `arguments` is:

```
arguments {context}
```

where:

`context` specifies the program context to display. If *`context`* is not specified, the current context is used (normally the procedure active when the program was suspended).

Usage

You use the `arguments` command to display the name and context of each argument within the specified context.

7.4.7 backtrace

The `backtrace` command prints information about all currently active procedures, starting with the most recent, or for a given number of levels.

Syntax

The syntax of `backtrace` is:

```
backtrace {count}
```

where:

`count` specifies the number of levels to trace. This is an optional argument. If you do not specify *`count`*, the currently active procedures are traced.

Usage

When your program has stopped running, because of a breakpoint or watchpoint, you use `backtrace` to extract information on currently active procedures. You can access information like the current function, the line of source code calling the function and so on.

7.4.8 break

The `break` command allows you to specify breakpoints.

Syntax

The syntax of the `break` command is:

```
break{/size} {loc {count} {do '{'command{;command}'}'} {if expr}}
```

where:

/size specifies which code type to break:

`/16` specifies the instruction size as Thumb.

`/32` specifies the instruction size as ARM.

With no *size* specifier, `break` tries to determine the size of breakpoint to use by extracting information from the nearest symbol at or below the address to be broken. This usually chooses the correct size, but you can set the size explicitly.

loc specifies where the breakpoint is to be inserted. For more information, see *Program locations* on page 7-50.

count specifies the number of times the statement must be executed before the program is suspended. It defaults to 1, so if *count* is not specified, the program will be suspended the first time the breakpoint is encountered.

`do` specifies commands to be executed when the breakpoint is reached. Note that these commands must be enclosed in braces, represented in the pattern above by braces within quotes. Each command must be separated by semicolons.

If you not specify a `do` clause, `break` displays the program and source line at the breakpoint. If you want the source line displayed in conjunction with the `do` clause, use `where` as the first command in the `do` clause to display the line.

expr makes the breakpoint conditional upon the value of *expr*.

Usage

The break `command` specifies breakpoints at:

- procedure entry and exit
- lines
- statements within a line.

Each breakpoint is given a number prefixed by #. A list of current breakpoints and their numbers is displayed if `break` is used without any arguments. If a breakpoint is set at a procedure exit, several breakpoints may be set, with one for each possible exit.

Note

Use `unbreak` to delete any unwanted breakpoints. This is described in *unbreak* on page 7-45.

7.4.9 call

The `call` command calls a procedure.

Syntax

The syntax of the `call` command is:

```
call {/size} loc {(expression- list)}
```

where:

/size specifies which code type to break:

`/16` specifies the instruction size as Thumb.

`/32` specifies the instruction size as ARM.

With no *size* specifier, `call` tries to determine the instruction set of the destination code by extracting information from the nearest symbol at or below the address to call. This usually chooses the correct size, but you can set the size explicitly. The command correctly sets the PSR T-bit before the call and restores it on exit.

loc is a function or low-level address.

expression_list

is a list of arguments to the procedure. String literals are not permitted as arguments. If you specify more than one expression, separate the expressions with commas.

Usage

If the procedure (or function) returns a value, examine it using:

`print $result` for integer variables

`print $fpresult` for floating-point variables.

7.4.10 coproc

The `coproc` command describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display.

Syntax

The syntax of the `coproc` command is:

```
coproc cpnum {rno{:rno1} size access values {displaydesc}*}*
```

where:

size is the register size in bytes.

access may comprise the letters:

- `R` the register is readable.
- `W` the register is writable.
- `D` the register is accessed through `CPDT` instructions (if not present, the register is accessed through `CPRT`s).

values the format depends on whether the register is to be accessed through `CPRT` instructions. If so, it comprises four integer values separated by a space or comma. These values form bits 0 to 7 and 16 to 23 of a `MRC` instruction to read the register, and bits 0 to 7 and 16 to 23 of a `MCR` instruction to write the register:

```
r0_7, r16_23, w0_7, w16_23
```

If not, it comprises two integer values to form bits 12 to 15 and bit 22 of `CPDT` instructions to read and write the register:

```
b12_15, b22
```

displaydesc is one of the items listed in Table 7-2 on page 7-18.

Usage

Each command may describe one register, or a range of registers, that are accessed and formatted uniformly.

Example

For example, the floating-point coprocessor might be described by the command:

```
copro 1 0:7 16 RWD 1,8
  8 4 RW 0x10,0x30,0x10,0x20 w0[16:20] ’izoux’ "_" w0[0:4] ’izoux’
   9 4 RW 0x10,0x50,0x10,0x40
```

Table 7-2 Values for displaydesc argument

Item	Definition		
`string`	is printed as is.		
`field string`	`string`	is to be used as a `printf` format string to display the value of `field`.	
	`field`	is one of the forms:	
		`wn`	the whole of the nth word of the register value
		`w[bit]`	bit `bit` of the nth word of the register value
		`wn[bit1:bit2]`	bits `bit1` to `bit2` inclusive of the *n*th word of the register value. The bits may be given in either order.
`field ’{’ string {string}* ’}’`	`field`	must take be either `wn[bit]` or `wn[bit1:bit2]`. There must be one string for each possible value of `field`. The string in the appropriate position for the value of `field` is displayed (the first string for value 0, and so on).	
`field ’letters’`	`field`	must take one of the forms `wn[bit]` or `wn[bit1:bit2]` above. There must be one character in `letters` for each bit of `field`. The letters are displayed in uppercase if the corresponding bit of the field is set, and in lowercase if it is clear. The first letter represents the lowest bit if `bit1` < `bit2`. Otherwise it represents the highest bit.	

7.4.11 context

The `context` command sets the context in which the variable lookup occurs.

Syntax

The syntax of the `context` command is:

```
context context
```

where:

`context` specifies the program context. If *`context`* is not specified, the context is reset to the active procedure.

Usage

The `context` command affects the default context used by commands which take a context as an argument. When program execution is suspended, the search context is set to the active procedure.

7.4.12 cregisters

The `cregisters` command displays the contents of all readable registers of a coprocessor.

Syntax

The syntax of the cregisters command is:

```
cregisters cpnum
```

where

`cpnum` selects the coprocessor.

Usage

The contents of the registers is displayed in the format specified by an earlier `coproc` command. The formatting options are described in Table 7-2 on page 7-18.

7.4.13 cregdef

The `cregdef` command describes how the contents of a coprocessor register are formatted for display.

Syntax

The syntax of the `cregdef` command is:

```
cregdef cpnum rno displaydesc
```

where:

cpnum selects the coprocessor.

rno selects the register number in the selected coprocessor.

displaydesc describes how the processor contents are formatted for display.

Usage

The contents of the registers is displayed according to the formatting options described in Table 7-2 on page 7-18.

7.4.14 cwrite

The `cwrite` command writes to a coprocessor register.

Syntax

The syntax of the `cwrite` command is:

```
cwrite cpnum rno val{val...}*
```

where:

cpnum selects the coprocessor.

rno selects the register number in the named coprocessor.

val each *val* is an integer value and there must be one *val* item for each word of the coprocessor register.

Usage

Before you write to a coprocessor register, you must define that register as writable. This is described in *coproc* on page 7-17.

7.4.15 examine

The `examine` command allows you to examine the contents of memory.

Syntax

The syntax of the `examine` command is:

```
examine {expression1} {, {+}expression2 }
```

where:

expression1 gives the start address. The default address used is either:

- the address associated with the current context, minus 64, if the context has changed since the last `examine` command was issued
- the address following the last address displayed by the last `examine` command, if the context has not changed since the last `examine` command was issued.

expression2 specifies the end address, which may take three forms:

- if omitted, the end address is the value of the start address +128
- if *expression2* is preceded by +, the end address is given by the value of the start line + *expression2*
- if there is no +, the end line is the value of *expression2.*

The `$examine_lines` variable can be used to alter the default number of lines displayed from its initial value of 8 (128 bytes).

Usage

This command allows you to examine the contents of the memory between a pair of addresses, displaying it in both hexadecimal and ASCII formats, with 16 bytes per line. Low-level symbols are accepted by default.

7.4.16 find

The `find` command finds all occurrences in memory of a given integer value or character string.

Syntax

The syntax of the find command is either of the following:

```
find expression1 {,expression2 {,expression3}}
```

```
find string {,expression2 {,expression3}}
```

where:

expression1 gives the words in memory to search for

expression2 specifies the lower boundary for the search. If *expression2* is absent, the base of the currently loaded image is used.

expression3 specifies the upper boundary for the search. If *expression3* is absent, the top (R/W limit) of the currently loaded image is used.

string specifies the string to search for.

Usage

If the first form is used, the search is for words in memory whose contents match the value of *expression1*.

If the second form is used, the search is for a sequence of bytes in memory (starting at any byte boundary) whose contents match those of *string*.

Low-level symbols are accepted by default.

7.4.17 fpregisters

The `fpregisters` command displays the contents of the eight floating-point registers f0 to f7 and the floating-point processor status register (FPSR).

Syntax

The syntax of the `fpregisters` command is:

```
fpregisters[/full]
```

where:

`/full` includes detailed information on the floating-point numbers in the registers.

Usage

There are two formats for the display of floating-point registers.

`fpregisters` displays the registers and FPSR, and the full version includes detailed information on the floating-point numbers in the registers. This produces the following display:

```
f0 = 0              f1 = 3.1415926535
f2 = Inf            f3 = 0
f4 = 3.1415926535   f5 = 1
f6 = 0              f7 = 0
fpsr = %IZOux_izoux
```

`fpregisters/full`

produces a more detailed display:

```
f0 = I + 0x3fff 1 0x0000000000000000
f1 = I + 0x4000 1 0x490fdaa208ba2000
f2 = I +u0x43ff 1 0x0000000000000000
f3 = I - 0x0000 0 0x0000000000000000
f4 = I + 0x4000 1 0x490fdaa208ba2000
f5 = I + 0x3fff 1 0x0000000000000000
f6 = I + 0x0000 0 0x0000000000000000
f7 = I + 0x0000 1 0x0000000000000000
fpsr = 0x01070000
```

(Note that `fpregisters/full` does not output both sets of values.)

The format of this display is (for example):

```
F S Exp     J Mantissa
I +u0x43ff  1 0x0000000000000000
```

where:

F is a precision/format specifier:

`F`	single precision
`D`	double precision
`E`	extended precision
`I`	internal format
`P`	packed decimal

S is the sign.

Exp is the exponent.

J is the bit to the left of the binary point.

Mantissa are the digits to the right of the binary point.

u The `u` between the sign and the exponent indicates that the number is flagged as *uncommon*, in this example infinity. This applies only to internal format numbers.

In the FPSR description, the first set of letters indicates the floating-point mask and the second the floating-point flags. The status of the floating-point mask and flag bits is indicated by their case. Uppercase means the flag is set and lowercase means that it is cleared.

The flags are:

`I`	Invalid operation
`Z`	Divide by zero
`O`	Overflow
`U`	Underflow
`X`	Inexact

7.4.18 go

The `go` command starts execution of the program.

Syntax

The syntax of the `go` command is:

```
go {while expression}
```

where:

`while` If `while` is used, *`expression`* is evaluated when a breakpoint is reached. If *`expression`* evaluates to true (that is. non-zero), the breakpoint is not reported and execution continues.

`expression` specifies the expression to evaluate.

Usage

The first time `go` is executed, the program starts from its normal entry point. Subsequent `go` commands resume execution from the point at which it was suspended.

7.4.19 getfile

The `getfile` command reads from a file and writes the content to memory.

Syntax

The syntax of the `getfile` command is:

```
getfile filename expression
```

where:

`filename` names the file to read from.

`expression` defines the memory location to write to.

Usage

The contents of the file are read as a sequence of bytes, starting at the address which is the value of *`expression`*. Low-level symbols are accepted by default.

7.4.20 help

The `help` command displays a list of available commands, or help on commands.

Syntax

The syntax of the `help` command is:

```
help {command}
```

where:

`command` is the name of the command you want help on.

Usage

The display includes syntax and a brief description of the purpose of each command. If you need information about all commands, as well as their names, type `help *`.

7.4.21 in

The `in` command changes the current context by one activation level.

Syntax

The syntax of the in command is:

```
in
```

Usage

The `in` command sets the context to that called from the current level. It is an error to issue an `in` command when no further movement in that direction is possible.

7.4.22 istep

The `istep` command steps execution through one or more instructions.

Syntax

The syntax of the `istep` command is:

```
istep {in} {count|w{hile} expression}
istep out
```

Usage

This command is analogous to the `step` command except that it steps through one instruction at a time, rather than one high-level language statement at a time.

7.4.23 language

The `language` command sets the high-level language.

Syntax

The syntax of the language command is:

```
language {language}
```

where:

`language` specifies the language to use. Enter one of the following:

- `none`
- `C`
- `F77`
- `PASCAL`
- `ASM`

Usage

The symbolic debugger uses any high-level debugging tables generated by a compiler to set the default language to the appropriate one for that compiler, whether it is Pascal, Fortran or C. If it does not find high-level tables, it sets the default language to none, and modifies the behavior of `where` and `step` so that:

`where` reports the current program counter and instruction

`step` steps by one instruction

7.4.24 let

The `let` command allows you to change the value of a variable or contents of a memory location.

Syntax

The syntax of the `let` command is:

{let} {*variable* | *location*} = *expression*{{,} *expression*}*

where:

variable names the variable to change.

location names the memory location to change.

expression contains the expression or expressions.

Usage

The `let` command is used in low-level debugging to change memory. If the left-side expression is a constant or a true expression (and not a variable), it is treated as a word address, and memory at that location (and if necessary the following locations) is changed to the values in the following expression(s).

An equals sign (=) or a colon (:) can separate the variable or location from the expression. If multiple expressions are used, they must be separated by commas or spaces.

Variables can only be changed to compatible types of expression. However, the debugger converts integers to floating-point and vice versa, rounding to zero. The value of an array can be changed, but not its address, because array names are constants. If the subscript is omitted, it defaults to zero.

If multiple expressions are specified, each expression is assigned to `variable[`*n*`-1]`, where *n* is the nth expression.

See also *let* on page 7-57 for more information on the `let` command.

Specifying the source directory

The variable `$sourcedir` is used to specify the directory or directories which contain the program source files. It can be set using the command:

```
{let} $sourcedir = string
```

where *`string`* should be a valid directory name. For example:

```
let $sourcedir=myhome
```

The debugger reassigns the value of `$sourcedir` and stores the two values *`dir`* and *`old_dir`*:

```
ARMSD: let $sourcedir = "temp"
ARMSD: p $sourcedir
"temp"
ARMSD: lo dhry
ARMSD: b main:119
ARMSD: go

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without ’register’ attribute
Breakpoint #1 at #dhry_1:main, line 119 of dhry_1.c
  119     printf ("\n");

ARMSD: p $sourcedir
"temp source_dir"
```

You can also include pathnames for directory trees, for example:

```
let $sourcedir=myhome src src/src2 src/src2/src3
```

Note

No warning is displayed if you enter an invalid file or pathname.

Command-line arguments

Command-line arguments for the debuggee can be specified using the `let` command with the root-level variable `$cmdline`. The syntax in this case is:

```
{let} $cmdline = string
```

The program name is automatically passed as the first argument, and thus should not be included in the string. The setting of `$cmdline` can be examined using `print`.

`go` starts execution of the program.

`getfile` reads the contents of an area of memory from a file.

`load` loads an image for debugging.

`putfile` writes the contents of an area of memory to a file.

`reload` reloads the object file specified on the armsd command line, or the last load command.

`type` types the contents of a source file, or any text file, between a specified pair of line numbers.

Reading and writing bytes and halfwords (shorts)

When you specify a write to memory in armsd, a word value is used. For example:

```
 let 0x8000 = 0x01
```

makes armsd transfer a word (4 bytes) to memory starting at the address 0x8000. The bytes at 0x8001, 0x8002 and 0x8003 are zeroed.

To write only a single byte, you must indicate that a byte transfer is required. You can do this with:

```
let *(char *)0xaddress = value
```

Similarly, to read from an address use:

```
print *(char *)0xaddress
```

You can also read and write half-words (shorts) in a similar way:

```
let *(short *)0x8000 = value
```

```
print /%x *(short *)0x8000
```

where `/%x` displays in hex.

7.4.25 list

The `list` command displays the contents of the memory between a specified pair of addresses in hexadecimal, ASCII and instruction format, with four bytes (one instruction) per line.

Syntax

The syntax of the `list` command is:

```
list{/size} {expression1}{, {+}expression2 }
```

where:

`size` distinguishes between ARM and Thumb code:

`/16` lists as Thumb code.

`/32` lists as ARM code.

With no `size` specifier, `list` tries to determine the instruction set of the destination code by extracting information from the nearest symbol at or below the address to start the listing.

`expression1` gives the start address. If unspecified, this defaults to either:

- the address associated with the current context minus 32, if the context has changed since the last `list` command was issued
- the address following the last address displayed by the last `list` command, if the context has not changed since the last `list` command was issued.

`expression2` gives the end address. It may take three forms:

- if *`expression2`* is omitted, the end address is the value of the start address + 64
- if it is preceded by +, the end address is the start line + *`expression2`*
- if there is no +, the end line is the value of *`expression2`*.

Usage

The `$list_lines` variable can alter the default number of lines displayed from its initial value of 16 (64 bytes).

Low-level symbols are accepted by default.

7.4.26 load

The `load` command loads an image for debugging.

Syntax

The syntax of the `load` command is:

```
load{/profile-option} image-file {arguments}
```

where:

`profile-option` specifies which profiling option to use:

`/callgraph` directs the debugger to provide the image being loaded with counts which enable the dynamic call-graph to be constructed (for use with profiling).

`/profile` directs the debugger to prepare the image being loaded for flat profiling.

`image-file` is the name of the file to be debugged.

`arguments` are the command-line arguments the program normally takes.

Usage

`image-file` and any necessary *`arguments`* may also be specified on the command-line when the debugger is invoked. See *Command-line options* on page 7-3 for more information.

If no arguments are supplied, the arguments used in the most recent load or reload, setting of `$cmdline`, or command-line invocation are used again.

The `load` command clears all breakpoints and watchpoints.

7.4.27 log

The `log` command sends the output of subsequent commands to a file as well as to the screen.

Syntax

The syntax of the `log` command is:

```
log filename
```

where:

`filename` is the name of the file where the record of activity is being stored.

Usage

To terminate logging, type `log` without an argument. The file can then be examined using a text editor or the `type` command.

Note

The debugger prompt and the debug program input/output is not logged.

7.4.28 lsym

The `lsym` command displays low-level symbols and their values.

Syntax

The syntax of the `lsym` command is:

`lsym` *`pattern`*

where:

`pattern` is a symbol name or part of a symbol name.

Usage

The wildcard (*) matches any number of characters at the start and/or end of the pattern:

`lsym *fred` displays information about fred, alfred

`lsym fred*` displays information about fred, frederick

`lsym *fred*` displays information about alfred, alfreda, fred, frederick

The wildcard `?` matches one character:

`lsym ??fred` matches Alfred

`lsym Jo?` matches Joe, Joy, and Jon

7.4.29 obey

The `obey` command executes a set of debugger commands which have previously been stored in a file, as if they were being typed at the keyboard.

Syntax

The syntax of the `obey` command is:

`obey` *`command-file`*

where:

`command-file` is the file containing the list of commands for execution.

Usage

You can store frequently-used command sequences in files, and call them using `obey`.

7.4.30 out

The `out` command changes the current context by one activation level and sets the context to be that of the caller of the current context.

Syntax

The syntax of the `out` command is:

```
out
```

Usage

It is an error to issue an `out` command when no further movement in that direction is possible.

7.4.31 pause

The `pause` command prompts you to press a key to continue.

Syntax

The syntax of the `pause` command is:

`pause` *`prompt-string`*

where:

`prompt-string` is a character string written to `stderr`.

Usage

Execution continues only after you press a key. If you press ESC while commands are being read from a file, the file is closed before execution continues.

7.4.32 print

The `print` command examines the contents of the debugged program's variables, or displays the result of arbitrary calculations involving variables and constants.

Syntax

The syntax of the `print` command is:

```
print{/format} expression
```

where:

/format selects a display format, as described in Table 7-1 on page 7-11. If no */format* string is entered, integer values default to the format described by the variable `$format`. Floating-point values use the default format string `%g`.

expression enters the expression for evaluation.

Usage

Pointer values are treated as integers, using a default fixed format `%.8x`, for example, 000100e4.

See also *print* on page 7-56 for more information on the `print` command.

7.4.33 profclear

The `profclear` command clears profiling counts.

Syntax

The syntax of the `profclear` command is:

```
profclear
```

Usage

For more information on the ARM profiler, refer to Chapter 8 *Toolkit Utilities*.

7.4.34 profoff

The `profoff` command stops the collection of profiling data.

Syntax

The syntax of the `profoff` command is:

```
profoff
```

Usage

For more information on the ARM profiler, refer to Chapter 8 *Toolkit Utilities*.

7.4.35 profon

The `profon` command starts the collection of profiling data.

Syntax

The syntax of the `profon` command is:

```
profon {interval}
```

where:

`interval` is the time between PC-sampling in microseconds.

Usage

Lower values have a higher performance overhead, and slow down execution, but higher values are not as accurate.

7.4.36 profwrite

The `profwrite` command writes profiling information to a file.

Syntax

The syntax of the `profwrite` command is:

```
profwrite {filename}
```

where:

`filename` is the name of the file to contain the profiling data.

Usage

The generated information can be viewed using the `armprof` utility. This is described in Chapter 8 *Toolkit Utilities*.

7.4.37 putfile

The `putfile` command writes the contents of an area of memory to a file. The data is written as a sequence of bytes.

Syntax

The syntax of the `putfile` command is:

```
putfile filename expression1, {+}expression2
```

where:

`filename` specifies the name of the file to write the data into.

`expression1` specifies the lower boundary of the area of memory to be written.

`expression2` specifies the upper boundary of the area of memory to be written.

Usage

The upper boundary of the memory area is defined as follows:

- if *`expression2`* is not preceded by a `+` character, the upper boundary of the memory area is the value of:

  `expression2` `- 1`

- if *`expression2`* is preceded by a `+` character, the upper boundary of the memory area is the value of:

  `expression1` `+` *`expression2`* `- 1.`

Low-level symbols are accepted by default.

7.4.38 quit

The `quit` command terminates the current armsd session.

Syntax

The syntax of the `quit` command is:

```
quit
```

Usage

This command also closes any open log or obey files.

7.4.39 readsyms

The `readsyms` command loads debug information from a specified file (like the `symbols` command).

Syntax

The syntax of the `readsyms` command is:

```
readsyms
```

Usage

The corresponding code must be present in another way (for example, via a `getfile`, or by being in ROM).

7.4.40 registers

The `registers` command displays the contents of ARM registers 0 to 14, the program counter, and the status flags contained in the processor status register.

Syntax

The syntax of the `registers` command is:

`registers {`*`mode`*`}`

where:

`mode` selects the registers to display. For a list of mode names, refer to *Predefined symbols* on page 7-60.

This option may also take the value `all`, where the contents of all registers of the current mode are displayed, together with all banked registers for other modes with the same address width.

Usage

If used with no arguments, or if *`mode`* is the current mode, the contents of all registers of the current mode are displayed. If the *`mode`* argument is specified, but is not the current mode, the contents of the banked registers for that mode are displayed.

A sample display produced by `registers` might look like this:

Example 7-1

```
R0 = 0x00000000     R1 = 0x00000001     R2 = 0x00000002     R3 = 0x00000003
R4 = 0x00000004     R5 = 0x00000005     R6 = 0x00000006     R7 = 0x00000007
R8 = 0x00000008     R9 = 0x00000009     R10= 0x0000000a     R11= 0x0000000b
R12= 0x0000000c     R13= 0x0000000d     R14= 0x0000000e
PC = 0x00008000     PSR= %NzcVIF_SVC26
```

7.4.41 reload

The `reload` command reloads the object file specified on the armsd command line, or with the last `load` command.

Syntax

The syntax of the `reload` command is:

```
reload{/profile-option} {arguments}
```

where

profile-option specifies which profiling option to use:

`/callgraph` tells the debugger to provide the image being loaded with counts to enable the dynamic call-graph to be constructed (for use with profiling).

`/profile` directs the debugger to prepare the image being loaded for flat profiling.

arguments are the command-line arguments the program normally takes. If no *arguments* are specified, the arguments used in the most recent load or reload setting of `$cmdline` or command-line invocation are used again.

Usage

Breakpoints (but not watchpoints) remain set after a `reload` command.

7.4.42 return

The `return` command returns to the caller of the current procedure, passing back a result where required.

Syntax

The syntax of the `return` command is:

```
return {expression}
```

where:

expression contains the expression to be evaluated.

Usage

You cannot specify the return of a literal compound data type such as an array or record using this command, but you can return the value of a variable, expression or compound type.

7.4.43 step

The `step` command steps execution through one or more program statements.

Syntax

The syntax of the `step` command is:

```
step {in} {out} {count|w{hile} expression} step out
```

where:

`in` continues single-stepping into procedure calls, so that each statement within a called procedure is single-stepped. If `in` is absent, each procedure call counts as a single statement and is executed without single stepping.

`out` steps out of a function to the line of originating code which immediately follows that function. This is useful when `step in` has been used too often.

`count` specifies the number of statements to be stepped through: if it is omitted only one statement will be executed.

`while` continues single-stepped execution until its *`expression`* evaluates as false (zero).

`expression` is evaluated after every step.

Usage

To step by instructions rather than statements:

- use the `istep` command
- or enter `language none`.

7.4.44 symbols

The `symbols` command lists all symbols defined in the given or current context, with their type information.

Syntax

The syntax of the `symbols` command is:

```
symbols {context}
```

where:

`context` defines the program context:

- To see global variables, define *`context`* as the filename with no path or extension.
- To see internal variables, use `symbols $`.

Usage

The information produced is listed in the form:

```
name type, storage-class
```

7.4.45 type

The `type` command types the contents of a source file, or any text file, between a specified pair of line numbers.

Syntax

The syntax of the `type` command is:

```
type {expression1} {, {{+}expression2} {,filename} }
```

where:

`expression1` gives the start line. If *`expression1`* is omitted, it defaults to:

- the source line associated with the current context minus 5, if the context has changed since the last `type` command was issued
- the line following the last line displayed with the `type` command, if the context has not changed.

`expression2` gives the end line, in one of three ways:

- if *`expression2`* is omitted, the end line is the start line +10
- if *`expression2`* is preceded by `+`, the end line is given by the value of the start line `+` *`expression2`*
- if there is no `+`, the end line is simply the value of *`expression2`*.

Usage

To look at a file other than that of the current context, specify the filename required and the locations within it.

To change the number of lines displayed from the default setting of 10, use the `$type_lines` variable.

7.4.46 unbreak

The `unbreak` command removes a breakpoint.

Syntax

The syntax of the `unbreak` command is:

```
unbreak {location | #breakpoint_num}
```

where:

`location` is a source code location.

`breakpoint_num` is the number of the breakpoint

Usage

If there is only one breakpoint, delete it using `unbreak` without any arguments.

Note

A breakpoint always keeps its assigned number. Breakpoints are not renumbered when another breakpoint is deleted, unless the deleted breakpoint was the last one set.

7.4.47 unwatch

The `unwatch` command clears a watchpoint.

```
unwatch
```

Syntax

The syntax of the `unwatch` command is:

```
unwatch {variable | #watchpoint_number}
```

where:

`variable` is a variable name.

`variable` is the number of a watchpoint (preceded by #) set using the `watch` command.

Usage

If only one watchpoint has been set, delete it using `unwatch` without any arguments.

7.4.48 variable

The `variable` command provides type and context information on the specified variable (or structure field).

Syntax

The syntax of the `variable` command is:

```
variable variable
```

where:

`variable` specifies the variable to examine.

Usage

`variable` can also return the type of an expression.

7.4.49 watch

The `watch` command sets a watchpoint on a variable.

Syntax

The syntax of the `watch` command is:

```
watch {variable}
```

where:

`variable` names the variable to watch.

Usage

If *`variable`* is not specified, a list of current watchpoints is displayed along with their numbers. When the variable is altered, program execution is suspended. As with `break` and `unbreak`, these numbers can subsequently be used to remove watchpoints.

Bitfields are not watchable.

If EmbeddedICE is available, ensure that watchpoints use hardware watchpoint registers to avoid any performance penalty.

Note

When using the C compiler, be aware that the code produced can use the same register to hold more than one variable if their lifetimes do not overlap. If the register variable you are investigating is no longer being used by the compiler, you may see a value pertaining to a completely different variable.

Adding watchpoints may make programs execute very slowly, because the value of variables has to be checked every time they could have been altered. It is more practical to set a breakpoint in the area of suspicion and set watchpoints once execution has stopped.

7.4.50 where

The `where` command prints the current context and shows the procedure name, line number in the file, filename and the line of code.

Syntax

The syntax of the `where` command is:

```
where {context}
```

where:

`context` specifies the program context to examine.

Usage

If a context is specified after the `where` command, the debugger displays the location of that context.

7.4.51 while

The `while` command is only useful at the end of an existing statement. You enter multi-statement lines by separating the statements with `;` characters.

Syntax

The syntax of the `while` command is:

`statement; {statement;} while` *`expression`*

where:

`expression` gives the expression to be evaluated.

Usage

Interpretation of the line continues until *`expression`* evaluates to false (zero).

7.5 Specifying source-level objects

This section gives information on variables, program locations, expressions and constants.

7.5.1 Variable names and context

You can usually just refer to variables by their names in the original source code. To print the value of a variable, type:

```
print variable
```

High-level languages

With structured high-level languages, variables defined in the current context can be accessed by giving their names. Other variables should be preceded by the context (for example. filename of the function) in which they are defined. This also gives access to variables that are not visible to the executing program at the point at which they are being examined. The syntax in this case is:

```
procedure:variable
```

Global variables

Global variables can be referenced by qualifying them with the module name or filename if there is likely be any ambiguity. For example, because the module name is the same as a procedure name, you should prefix the filename or module name with #. The syntax in this case is:

```
#module:variable
```

Ambiguous declarations

If a variable is declared more than once within the same procedure, resolve the ambiguity by qualifying the reference with the line number in which the variable is declared as well as, or instead of, the function name:

```
#module:procedure:line-no:variable
```

Variables within activations of a function

To pick out a particular activation of a repeated or recursive function call, prefix the variable name with a backslash (\) followed by an integer. Use 1 for the first activation, 2 for the second and so on. A negative number will look backwards through activations of the function, starting with \-1 for the previous one. If no number is specified and multiple activations of a function are present, the debugger always looks at the most recent activation.

To refer to a variable within a particular activation of a function, use:

```
procedure\{-}activation-number:variable
```

Expressing context

The complete syntax for the various ways of expressing context is:

```
{#}module{{:procedure}*
{\{-}activation-number}}
{#}procedure{{:procedure}*
{\{-}activation-number}}
#
```

Specifying variable names

The complete syntax for specifying a variable name is:

```
{context:.{line-number:::}}variable
```

The various syntax extensions needed to differentiate between different objects rarely need to be used together.

7.5.2 Program locations

Some commands require arguments that refer to locations in the program. You can refer to a location in the program by:

- procedure entry and exit
- program line numbers
- statement within a line.

In addition to the high-level program locations described here, low-level locations can also be specified. See *Low-level symbols* on page 7-59 for further details.

Procedure entry and exit

Using a procedure name alone sets a breakpoint (see *break* on page 7-15) at the entry point of that procedure. To set a breakpoint at the end of a procedure, just before it returns, use the syntax:

```
procedure:$exit
```

Program line numbers

Program line numbers can be qualified in the same way as variable names, for example:

```
#module:123
procedure:3
```

Line numbers can sometimes be ambiguous, for example when a file is included within a function. To resolve any ambiguities, add the name of the file or module in which the line occurs in parentheses. The syntax is:

```
number(filename)
```

Statement within a line

To refer to a statement within a line, use the line number followed by the number of the statement within the line, in the form:

```
line-number.statement-number
```

So, for example, `100.3` refers to the third statement in line 100.

7.5.3 Expressions

Some debugger commands require expressions as arguments. Their syntax is based on C. A full set of operators is available. The lower the number, the higher the precedence of the operator. These are shown in the following table, in descending order of precedence.

Table 7-3 Precedence of operators

Precedence	Operator	Purpose	Syntax
1	`()`	Grouping	`a * (b + c)`
	`[]`	Subscript	`isprime[n]`
	`.`	Record selection	`rec.field,a.b.c`

Table 7-3 Precedence of operators (Continued)

Precedence	Operator	Purpose	Syntax
`rec->next`	`->`	Indirect selection	`rec->next` is identical to `(*rec).next`
2	`!`	Logical NOT	`!finished`
	`~`	Bitwise NOT	`~mask`
	`-`	Unary minus	`-a`
	`*`	Indirection	`*ptr`
	`&`	Address	`&var`
3	`*`	Multiplication	`a * b`
	`/`	Division	`a / b`
	`%`	Integer remainder	`a % b`
4	`+`	Addition	`a + b`
	`-`	Subtraction	`a - b`
5	`>>`	Right shift	`a >> 2`
	`<<`	Left shift	`a >> 2`
6	`<`	Less than	`a < b`
	`>`	Greater than	`a > b`
	`<=`	Less than or equal	`a <= b`
	`>=`	Greater than or equal	`a >= b`
7	`==`	Equal	`a == 0`
	`!=`	Not equal	`a != 0`
8	`&`	Bitwise AND	`a & b`
9	`^`	Bitwise EOR	`a ^ b`
10	`|`	Bitwise OR	`a | b`
11	`&&`	Logical AND	`a && b`
12	`||`	Logical OR	`a || b`

Subscripting can only be applied to pointers and array names. The symbolic debugger checks both the number of subscripts and their bounds, in languages which support such checking. It is inadvisable to use out-of-bound array accesses. As in C, the name of an array may be used without subscripting to yield the address of the first element.

The prefix indirection operator `*` is used to de-reference pointer values. If `ptr` is a pointer, `*ptr` yields the object to which it points.

If the left-hand operand of a right shift is a signed variable, the shift is an arithmetic one and the sign bit is preserved. If the operand is unsigned, the shift is a logical one and zero is shifted into the most significant bit.

Note

Expressions must not contain function calls that return non-primitive values.

7.5.4 Constants

Constants may be decimal integers, floating-point numbers, octal integers or hexadecimal integers. Note that `1` is an integer whereas `1.` is a floating-point number.

Character constants are also allowed. For example, `A` yields 65, the ASCII code for the character A.

Address constants may be specified by the address preceded with an `@` symbol. For commands which accept low-level symbols by default, the `@` may be omitted.

7.6 Armsd variables

This section lists the variables available in armsd, and gives information on manipulating them.

7.6.1 Summary of armsd variables

Many of the debugger's defaults can be modified by setting variables. Table 7-4 lists the variables. Most of these are described elsewhere in this chapter in more detail.

Table 7-4 armsd variables

Variable	Description
`$clock`	number of microseconds since simulation started. This variable is read-only. This variable is read only, and is only available if a processor clock speed has been specified (See the *ARM Software Development Toolkit User Guide* for information on how to specify the emulated processor clock speed)
`$cmdline`	argument string for the debuggee.
`$echo`	non-zero if commands from obeyed files should be echoed (initially set to 01).
`$examine_lines`	default number of lines for the `examine` command (initially set to 8).
`$int_format`	default format for printing integer values (initially set to %ld").
`$float_format`	default format for printing floating-point values (initially set to %ld").
`$fpresult`	floating-point value returned by last called function (junk if none, or if a floating-point value was not returned). This variable is read-only. `$fpresult` returns a result only if the image has been build for hardware floating-point. If the image is built for software floating-point, it returns zero.
`$inputbase`	base for input of integer constants (initially set to 10).
`$list_lines`	default number of lines for list command (initially set to 16).
`$memory_statistics`	outputs any memory map statistics which the ARMulator has been keeping. This variable is read-only. See the *ARM Software Development Toolkit User Guide* for further details.
`$rdi_log`	rdi logging is enabled if non-zero, and serial line logging is enabled if bit 1 is set (initially set to 0).

Table 7-4 armsd variables (Continued)

Variable	Description
`$result`	integer result returned by last called function (junk if none, or if an integer result was not returned). This variable is read-only.
`$sourcedir`	directory containing source code for the program being debugged (initially set to the current directory).
`$statistics`	outputs any statistics which the ARMulator has been keeping. This variable is read-only.
`$statistics_inc`	similar to `$statistics`, but outputs the difference between the current statistics and those when `$statistics` was last read. This variable is read-only.
`$top_of_memory`	This is used to enable EmbeddedICE to return sensible values when a HEAP_INFO SWI call is made to determine where the heap and stack should be placed in memory. The default is 0x80000 (512KB). This should be modified before executing a program on the target if the memory size available differs from this.
`$type_lines`	default number of lines for the `type` command.
`$vector_catch`	indicates whether or not execution should be caught when various conditions arise. The default value is `%RUsPDAifE`. Capital letters indicate that the condition is to be intercepted: `R` reset `U` undefined instruction `S` SWI `P` prefetch abort `D` data abort `A` address exception `I` IRQ `F` FIQ `E` Error (for Demon only. Do not use for Angel)

armsd internal variables

The variables in Table 7-5 are included to support EmbeddedICE and Multi-ICE.

Table 7-5 armsd variables for EmbeddedICE

Variable	Description
`$icebreaker_lockedpoints`	shows or sets locked EmbeddedICE macrocell points.
`$semihosting_enabled`	enables or disables semihosting.
`$semihosting_vector`	sets up semihosting SWI vector. Semihosting is described in the *ARM Software Development Toolkit User Guide*.
`$semihosting_arm_swi`	defines which ARM SWIs are interpreted as semihosting requests by the debug agent.
`$semihosting_thumb_swi`	defines which Thumb SWIs are interpreted as semihosting requests by the debug agent.

7.6.2 Accessing variables

The following commands are available for accessing variables.

print

This command examines the contents of the debugged program's variables, or displays the result of arbitrary calculations involving variables and constants. Its syntax is:

```
p{rint}{/format} expression
```

For example:

```
print/%x listp->next
```

prints field `next` of structure `listp`.

If no format string is entered, integer values default to the format described by the variable `$format`. Floating-point values use the default format string `%g`. Pointer values are treated as integers, using a default fixed format `%.8x`, for example, 000100e4.

let

The `let` command allows you to change the value of a variable or contents of a memory location. Its syntax is:

```
{let} variable = expression{{,} expression}*
{let} memory-location = expression{{,} expression}*
```

An equals sign(=) or a colon(:) can be used to separate the variable or location from the expression. If multiple expressions are used, they must be separated by commas or spaces.

Variables can only be changed to compatible types of expression. However, the debugger will convert integers to floating-point and vice versa, rounding to zero. The value of an array can be changed, but not its address, because array names are constants. If the subscript is omitted, it defaults to zero. If multiple expressions are specified, each expression is assigned to `variable[` *n*`-1]`, where *n* is the nth expression.

The `let` command is used in low-level debugging to change memory. If the left-hand side expression is a constant or a true expression (and not a variable) it is treated as a word address, and memory at that location (and if necessary the following locations) is changed to the values in the following expression(s).

Available variable formats are now:

- `$format_int`
- `$format_uint`
- `$format_float`
- `$format_sbyte`
- `$format_ubyte`
- `$format_string`
- `$format_complex.`

7.6.3 Formatting integer results

You can set the default format string used by the `print` command for the output of integer results by using `let` with the root-level variable `$format`. This is initially set to `%d`.

```
{let} $format = string
```

Note

When using floating-point formats, integers will not print correctly. The contents of *`string`* should be a format as described in *Names used in syntax descriptions* on page 7-10.

7.6.4 Specifying the base for input of integer constants

You use the `$inputbase` variable to set the base used for the input of integer constants.

```
{let} $inputbase = expression
```

If the input base is set to 0, numbers will be interpreted as octal if they begin with 0. Regardless of the setting of `$inputbase`, hexadecimal constants are recognized if they begin with 0x.

Note

`$inputbase` only specifies the base for the input of numbers. Specify the output format by setting `$format` to an appropriate value.

7.7 Low-level debugging

Low-level debugging tables are generated automatically when programs are linked with the `-debug` flag set (this is enabled by default). You cannot include high-level debugging tables in an image without also including low-level debugging tables. There is no need to enable debugging at the compilation stage for low-level debugging only; you just specify debugging when linking the program.

7.7.1 Low-level symbols

Low-level symbols are differentiated from high-level ones by preceding them with `@`.

The differences between high and low-level symbols are:

- a low-level symbol for a procedure refers to its call address, often the first instruction of the stack frame initialization
- the corresponding high-level symbol refers to the address of the code generated by the first statement in the procedure.

Low-level symbols can be used with most debugger commands. For example, when used with the `watch` command they stop execution if the word at the location named by the symbol changes. Low-level symbols can also be used where a command would expect an address expression.

Certain commands (`list`, `find`, `examine`, `putfile`, and `getfile`) accept low-level symbols by default. To specify a high-level symbol, precede it by `^`.

Memory addresses can also be used with commands and should also be preceded by `@`.

Note

Low-level symbols do not have a context and so they are always available.

7.7.2 Predefined symbols

There are several predefined symbols, as shown in Table 7-6. To differentiate these from any high-level or low-level symbols in the debugging tables, precede them with #.

Table 7-6 High-level symbols for low-level entities

Symbol	Description
r0 - r14	The general-purpose ARM registers 0 to 14.
r15	The address of the instruction which is about to execute. This may include the condition code flags, interrupt enable flags, and processor mode bits, depending on the target ARM architecture. Note that this value may be different from the real value of register 15 due to the effect of pipelining.
pc	The address of the instruction which is about to execute.
sp	The stack pointer (r13).
lr	The link register (r14)
fp	The frame pointer (r11).
psr and cpsr	psr and cpsr are synonyms for the current mode's processor status register. The values displayed for the condition code flags, interrupt enable flags, and processor mode bits, are an alphabetic letter per condition code and interrupt enable flag, and a mode name (preceded by an underscore) for the mode bits. This mode name will be one of USER26, IRQ26, FIQ26, SVC26, USER32, IRQ32, FIQ32, SVC32, UNDEF32, ABORT32 and SYSTEM32. Note that spsr is not defined if the processor is not capable of 32-bit operation. See also *Application Note 11, Differences Between ARM6 Series and Earlier Processors*.
spsr	spsr is the saved status register for the current mode. The values displayed are listed above in psr and cpsr.
f0 to f7	The floating-point registers 0 to 7.
fpsr	The floating-point status register.
fpcr	The floating-point control register.
a1 to a4	These refer to arguments 1 to 4 in a procedure call (stored in r0 to r3).
v1 to v7	These refer to the five to seven general-purpose register variables which the compiler allocates (stored in r4 to r10).

Table 7-6 High-level symbols for low-level entities (Continued)

Symbol	Description
`sb`	Static base, as used in re-entrant variants of the ARM Procedure Call Standard (APCS) (r9/v6).
`sl`	The stack limit register, used in variants of the APCS which implement software stack limit checking (r10/v7).
`ip`	Used in procedure entry and exit and as a scratch register (r12).

Printing register information

All these registers can be examined with the `print` command and changed with the `let` command. For example, the following form displays the processor status register (psr):

```
print/%x #psr
```

Setting the PSR

The `let` command can also set the psr, using the usual syntax for psr flags.

For example, the `N` and `F` flags could be set, the `V` flag cleared, and the `I`, `Z` and `C` flags left untouched and the processor set to 32-bit supervisor mode, by typing:

```
let #psr = %NvF_SVC32
```

The following example changes to User mode:

```
psr = %_User32
```

Note

The percentage sign must precede the condition flags and the underscore which in turn must precede the processor mode description.

Using # with low-level symbols

Normally, you do not need to use # to access a low-level symbol. You can use # to force a reference to a root context if you see the error message:

```
Error: Name not found
```

For example, use `#pc=0` instead of `pc=0`.

7.8 armsd commands for EmbeddedICE

The following armsd commands are included for compatibility with EmbeddedICE.

7.8.1 listconfig

The `listconfig` command lists the configurations known to the debug agent.

Syntax

The syntax of the `listconfig` command is:

```
listconfig file
```

where:

`file` specifies the file where the list of configurations is written.

7.8.2 loadagent

The `loadagent` command downloads a replacement EmbeddedICE ROM image, and starts it (in RAM).

Syntax

The syntax of the `loadagent` command is:

```
loadagent
```

7.8.3 loadconfig

The `loadconfig` command loads an EmbeddedICE configuration data file.

Syntax

The syntax of the `loadconfig` command is:

```
loadconfig file
```

where:

`file` names the EmbeddedICE configuration data file to load.

7.8.4 readsyms

The `readsyms` command loads an image file containing debug information but does not download the image.

Syntax

The syntax of the `readsyms` command is:

```
readsyms file
```

where:

`file` names the image file to load.

7.8.5 selectconfig

The `selectconfig` command selects the EmbeddedICE configuration to use.

Syntax

The syntax of the `selectconfig` command is:

```
selectconfig name version
```

where:

`name` is the name of the configuration data to be used

`version` indicates the version which should be used:

- `any` accepts any version number. This is the default.
- *n* uses version *n*.
- *n*+ uses version *n* or later.

7.9 Accessing the Debug Communications Channel

The debugger accesses the debug communication channel using the following commands.

For more information, see *Application Note 38: Using the ARM7TDMI's Debug Communication Channel.*

7.9.1 ccin

The `ccin` command selects a file containing Communications Channel data for reading. This command also enables Host to Target Communications Channel communication.

Syntax

The syntax of the `ccin` command is:

```
ccin filename
```

where:

`filename` names the file containing the data for reading.

7.9.2 ccout

The ccout command selects a file where Communications Channel data is written, and also enables Target to Host Communications Channel communication.

Syntax

The syntax of the ccout command is:

```
ccout filename
```

where:

`filename` names the file where the data is written.

Chapter 8
Toolkit Utilities

This chapter describes the software utilities provided with the Software Development Toolkit. It contains the following sections:

8.1 Functions of the toolkit utilities

fromelf The fromELF utility takes linker output files and produces image files. You can use it to produce any of a variety of formats of image file. You can also use it to display various information from the input files, or to generate text files containing the information.

armprof The ARM profiler displays an execution profile of a program from a profile data file generated by an ARM debugger.

armlib The ARM librarian enables sets of AOF files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several AOF files.

decaof The ARM Object Format decoder decodes AOF files such as those produced by armasm and armcc.

decaxf The ARM Executable Format decoder decodes executable files such as those produced by armlink or armcc.

topcc The ANSI to PCC C Translator helps to translate C programs and headers from ANSI C into PCC C, primarily by rewriting top-level function prototypes.

topcc is available for UNIX platforms only, not for Windows.

Flash downloader The Flash downloader enables you to download binary images to the Flash memory of supported ARM development and evaluation boards.

8.2 The fromELF utility

fromELF is a utility that can translate *Executable Linkable Format* (ELF) image format files produced by the ARM Linker into other formats.

fromELF outputs the following image formats:

- ARM Image Format (AIF) family. The AIF family includes executable AIF, and non-executable AIF.
- Plain binary format.
- Extended Intellec Hex (IHF) format.
- Motorola 32 bit S record format.
- Intel Hex 32 format.
- ELF format (resaves as ELF).

8.2.1 fromELF command-line options

The `fromELF` command syntax is as follows:

```
fromelf  [-h] [-nodebug] [-nozeropad] input_file [-text/n…
[output_file]] [output_format output_file]…
```

where:

`-h` shows help and usage information. If this option is specified, fromELF exits after showing the help message. It ignores everything else on the command line. Calling fromELF without any parameters produces the same help information.

`-nodebug` does not put debug information in the output files. If `-nodebug` is specified, it affects all output formats. It overrides the `-text/g` option.

`-nozeropad` does not expand zero initialized areas in the output image. This option is valid only if the output is a binary file. It is ignored for all other file formats.

`input_file` specifies the ELF file to be translated. Pathnames with appropriate directory separators for the OS are accepted.

fromELF only accepts ARM executable ELF files.

fromELF does not accept linkable or relocatable ELF files.

`-text/`*`n...`* Displays text information, where *n...* selects any combination of the following information categories:

`c`	disassembles code
`d`	prints contents of the data sections
`g`	prints debug information
`r`	prints relocation information
`s`	prints the symbol table
`t`	prints the string table(s)
`v`	prints detailed information on each segment and section header of the image.

The information is displayed on stdout, if an *`output_file`* is not specified.

`output_format` can be one of the following:

`-aif`	*ARM Image Format* (AIF) file
`-bin`	plain binary
`-aifbin`	non-executable AIF
`-ihf`	Extended Intellec Hex format
`-m32`	Motorola 32-Bit format (32 bit ’S’ records)
`-i32`	Intel Hex-32 format
`-elf`	ELF format (resaves as ELF).

Each *`output_format`* must be followed by an *`output_file`*.

`output_file` specifies the name of the output file. Pathnames with appropriate directory separators for the OS are accepted.

Each *`output_file`* must be distinct. FromELF does not check if the same output file is specified for more than one output format, and simply uses the same filename to produce the output. If any file of the same name already exists, it is overwritten without any warning message.

If *`output_file`* is the same as *`input_file`*, fromELF does not create the output in that format, and moves on to the next format. A warning is given for each invalid output filename.

8.2.2 Multiple output formats

Although multiple output formats are allowed, multiple outputs using the same format are not allowed. Each *`output_format`* can only appear once in the list of outputs, and only one output file is written using any particular format.

The following command is invalid:

```
fromelf -nodebug inp.axf -aif out1.aif -bin out2.bin -aif out3.aif
```

The second use of `-aif` in the command list is ignored, and only two output files are created, `out1.aif`, and `out2.bin`.

8.2.3 Image structure

fromELF can translate a file from ELF to other formats. It cannot alter the image structure. It is not possible to change a scatter loaded ELF image into a non scatter loaded image in another format. Any structural information must be provided to the Linker at link time.

8.3 ARM profiler

The ARM Profiler, armprof, displays an execution profile of a program from a profile data file generated by an ARM debugger. The profiler displays one of two types of execution profile depending on the amount of information present in the profile data:

- If only pc-sampling information is present, the profiler can display only a flat profile giving the percentage time spent in each function itself, excluding the time spent in any of its children.
- If function call count information is present, the profiler can display a *call graph* profile that shows not only the percentage time spent in each function, but also the percentage time accounted for by calls to all children of each function, and the percentage time allocated to calls from different parents.

See *Profiling* on page 11-20 in the *ARM Software Development Toolkit User Guide* for additional information.

8.3.1 Profiler command-line options

A number of options are available to control the format and amount of detail present in the profiler output. The command syntax is as follows:

```
armprof [-parent | -noparent] [-child | -nochild] [-sort options]
prf_file
```

where:

`-parent` displays information about the parents of each function in the profile listing. This gives information about how much time is spent in each function servicing calls from each of its parents.

`-noparent` turns off the parent listing.

`-child` displays information about the children of each function. The profiler displays the amount of time spent by each child performing services on behalf of the parent.

`-nochild` turns off the child listing.

`-sort` *`option`* sorts the profile information using one of the following options:

`cumulative` sorts the output by the total time spent in each function and all its children.

`self` sorts the output by the time spent in each function (excluding the time spent in its children).

`descendants` sorts the output by the time spent in all a function's children but excluding time spent in the function itself.

`calls` sorts the output by the number of calls to each function in the listing.

`prf_file` specifies the file containing the profile information.

By default, child functions are listed, but not parent functions, and the output is sorted by cumulative time.

Example

```
armprof -parent sort.prf
```

8.3.2 Sample output

The profiler output is split into a number of sections, each section separated by a line. Each section gives information on a single function. In a flat profile (one with no parent or child function information) each section is a single line.

The following shows sample sections for functions called `insert_sort` and `strcmp`.

```
Name                cum%   self%        desc%        calls
----------------------------------------------------------------
       main                17.69%       60.06%       1
insert_sort         77.76% 17.69%       60.06%       1
       strcmp              60.06%       0.00%        243432
----------------------------------------------------------------
       qs_string_compare   3.21%        0.00%        13021
       shell_sort          3.46%        0.00%        14059
       insert_sort         60.06%       0.00%        243432
strcmp              66.75% 66.75%       0.00%        270512
----------------------------------------------------------------
```

8.4 ARM librarian

The ARM librarian, armlib, enables sets of AOF files to be collected together and maintained in libraries. Such a library can then be passed to the linker in place of several AOF files. However, linking with an object library file does not necessarily produce the same results as linking with all the object files collected into the object library file. This is because of the way armlink processes its input files:

- each object file in the input list appears in the output unconditionally (although unused areas are eliminated if the output is AIF or if the `-remove` option is specified)
- a module from a library file is only included in the output if an object file or previously processed library file refers to it.

For more information on how armlink processes its input files, refer to *Chapter 6 Linker*.

When an object is extracted from a library, armlib drops the path and drive components of the object name. When an object is added to a library, armlib always drops the drive component, and by default also drops the path component of the object name.

8.4.1 Librarian command-line options

The syntax of the armlib command is:

```
armlib [-h] [-c | -i | -d | -e] [-o] [-n] [-p] [-l] [-s] [-t dir]
[-v file] library [file_list | member_list]
```

where:

`-h or -help` gives online details of the armlib command.

`-c` creates a new library containing files in *`file_list`*.

`-i` inserts files in *`file_list`* into the library. Existing members of the library are replaced by members of the same name.

`-d` deletes members in *`member_list`* from the library.

`-e` extracts members in *`member_list`*, placing them in files of the same name. The contents of the library are not changed.

`-o` adds an external symbol table to an object library.This is on by default, but can be turned off by `-n`.

`-n` does not add an external symbol table to an object library.

`-p` respects paths of files and objects. Note that any drive names specified are always dropped.

`-l` lists the library. This may be specified together with any other option.

`-s` lists the symbol table. This may be specified together with any other option

`-t` *`dir`* extracts files to a different directory, specified in *`dir`*. This option is used in conjunction with `-e`.

`-v` *`file`* reads in additional arguments from a file, in the same way as the armlink `-via` option, described in Chapter 6 *Linker*.

`library` names the library file.

`file_list` names the files to be used as input to the library. You can use the wildcards `*` and `?` to specify the files.

`member_list` names the library members to work on. You can use the wildcards `*` and `?` to specify the library members.

8.4.2 Examples

```
armlib -c mylib obj1 obj2 obj3...
armlib -e mylib ?sort*
armlib -e -t /tmp/mylib mylib.a
armlib -d mylib hash
armlib -i mylib quick_sort.o quick_hash1.o
armlib -l -s ansilib
```

8.5 ARM object file decoder

The ARM Object Format file decoder, decaof, is a tool that decodes AOF files such as those produced by armasm and armcc. The full specification of AOF can be found in Chapter 14 *ARM Object Library Format*.

8.5.1 Object file decoder command-line options

The syntax of the `decaof` command is:

```
decaof [-a] [-b] [-c] [-d] [-g] [-h] [-q] [-r] [-s] [-t] [-z] file
[file...]
```

where:

`-a` prints area contents in hex (and implicitly includes `-d`).

`-b` prints only the area declarations (brief).

`-c` disassembles code areas (and implicitly includes `-d`).

`-d` prints area declarations.

`-g` prints debug areas formatted readably.

`-h` gives online details of the decaof command.

`-q` gives a summary of the area sizes only.

`-r` prints relocation directives (and implicitly includes `-d`).

`-s` prints the symbol table.

`-t` prints the string table(s).

`-z` prints a one-line code and data size summary per file.

`file` names the AOF file(s) to decode. Each file must be an AOF file.

If no options are specified, the effect is of `-dst`.

8.5.2 Example

```
decaof -q test.o
       C$$code  4748
       C$$data  152
```

8.6 ARM executable format decoder

The ARM executable format file decoder, decaxf, is a tool that decodes executable files such as those produced by armlink.

8.6.1 Executable file decoder command-line options

The syntax of the `decaxf` command is:

```
decaxf [-c] [-g] [-h] [-s] [-t] [-d] [-r] file [file...]
```

where:

`-c` disassembles code.

`-g` prints debug areas.

`-h` gives online details of the `decaxf` command. You can also enter `-help`.

`-s` prints the symbol table.

`-t` prints the string table(s) (for ELF files only)

`-r` displays relocation information (for ELF files only)

`-d` displays the contents of the data section(s) (for ELF files only)

`file` names the AXF file(s) to decode. Each file must be an AXF file.

If no options are specified, summary information about the segments of the file is output.

8.6.2 Examples

```
decaxf -gst my_elf.axf
decaxf -c test1.axf test2.axf test3.axf
```

8.7 ANSI to PCC C Translator

The topcc program helps to translate (suitable) C programs and headers from the ANSI dialect of C into the PCC dialect of C. It does this mainly by rewriting top-level function prototypes (whether declarations or definitions).

The topcc translator does its translation before the C preprocessing phase of any following compilation, and ignores preprocessor flag settings. It cannot help with the translation of sources if function prototypes have been obscured, for example, by preprocessor macros.

The translation is limited, and other differences between the ANSI and PCC dialects must be dealt with in the source before translation, although some can be corrected after translation.

topcc is available for UNIX platforms only, not for Windows.

8.7.1 ANSI to PCC C command-line options

The command syntax for topcc is:

```
topcc [-d] [-c] [-e] [-p] [-s] [-t] [-v] [-l] [input_file
[output_file]]
```

where:

`-d`	describes what the program does
`-c`	does not remove the keyword **`const`**
`-e`	does not remove `#error....`
`-p`	does not remove `#pragma....`
`-s`	does not remove the keyword **`signed`**
`-t`	does not remove the second argument to `va_start()`
`-v`	does not remove the keyword **`volatile`**
`-l`	does not add `#line` directives
`input_file`	defaults to stdin
`output_file`	defaults to stdout

8.7.2 Translation details

Top-level function declarations are rewritten with their argument lists enclosed in `/*` and `*/`. For example, declarations like:

```
type foo(argument-list);
```

are rewritten as:

```
type foo(/* argument-list */);
```

Any comment tokens `/*` or `*/` in the original argument list are removed.

Function definition prototypes are rewritten in PCC style. For example, definitions like:

```
type foo(type1 a1, type2 a2) {...}
```

are rewritten as:

```
type foo(a1, a2)
type1 a1;
type2 a2;
{...}
```

and:

```
type foo(void)
{...
```

is rewritten as:

```
type foo()
{...
```

Notes on the example

The `...` declaration in a function definition (denoting a variable-length argument) is replaced by `int va_alist`. The second argument to calls of the `va_start` macro is removed. (`varargs.h` defines `va_start` as a macro taking one argument. `stdarg.h` adds a second argument.) However, topcc does not replace `#include <varargs.h>` with `#include <stdargs.h>`.

Warnings are given of occurrences of ANSI keywords **`const`**, **`enum`**, **`signed`**, and **`volatile`**.

`const`, **`signed`**, and **`volatile`** are removed.

Type `void *` is converted to `VoidStar`. To be compatible with PCC, this should be defined as `char *` using a **`typedef`**.

ANSI C's unsigned and unsigned long constants are rewritten using the typecasts `(unsigned)` and `(unsigned long)`. For example, `300ul` becomes `(unsigned long)300L`.

After rewrites that change the number of lines in the file, `#line` directives are included to resynchronize line numbering. These quote the source filename, so that debugging tools then refer to the ANSI form of sources.

8.7.3 Issues with topcc

topcc takes no account of the setting of conditional compilation options. This is deliberate. It converts all conditionally compilable variants in parallel.

- Braces must be nested reasonably within conditionally-compilable sections, or topcc may lose track of the brace nesting depth. This is used to determine whether it is within, or between, top-level definitions and declarations.
- It is not possible, in practice, to track brace-nesting depth without regard to reprocessing, as topcc uses heuristics to match conditionally-compiled braces. If topcc cannot match braces, it gives the message:

  ```
  mis-matched, conditionally included braces.
  ```

topcc cannot concatenate adjacent string literals. You can eliminate these from the input program beforehand. All important uses of ANSI-style implicit concatenation involve some mix of literals and preprocessor variables. topcc does not know about preprocessor variables.

If topcc finds an extra closing brace and starts processing text prematurely as if it were at the top level, it can damage function calls and macro invocations. In general, you should compare the output of topcc with its input (using a file difference utility) to check that changes have been reasonably localized to function headers and declarations. If necessary, you can inhibit most other translations to make these principal changes more visible. (See *ANSI to PCC C command-line options* on page 8-12)

8.8 The Flash downloader

The Flash downloader is a utility provided with the ARM Software Development Toolkit, and integrated into the ARM Debugger for Windows. The Flash downloader is installed in:

`ARM250\Bin\flash.li`, a little-endian version.

`ARM250\Bin\flash.bi`, a big-endian version.

8.8.1 The Flash downloader

You can use the Flash downloader to program Flash memory on the board. This works only if Angel is running from RAM (the default) at the time, or if EmbeddedICE interface or Multi-ICE is being used rather than Angel. The correct version for the endianness of the board should be used.

The downloaded file must be in plain binary format. Refer to *The fromELF utility* on page 8-3 for information on converting an ELF format file to plain binary.

The Flash downloader fails if it does not recognize the Flash being used, because it must understand how to program it. As supplied, the Flash downloader recognizes the two Flash devices supported by the ARM Development Board. Refer to the *ARM Target Development System User Guide* for more information.

If you are targeting a different target system you must produce your own download utility. You can use the source for the Flash downloader as a basis. The source is provided with the ARM Development Board, and is available from the ARM web site at `http://www.arm.com`.

If you are using armsd, the Flash downloader should be passed either the argument `-e`, or the name of the file to be downloaded into Flash.

If the file is being downloaded, you are prompted for the sector from where the programming should start. If you are downloading Angel, it should be programmed into the start of the Flash, from sector 0.

If you have the Angel Ethernet Kit, the Flash downloader program can be used to override the default IP address and net mask used by Angel for Ethernet communication. To do this from armsd, pass the Flash download program the argument `-e`. The program prompts for the IP address and net mask. If you are using the ARM Debugger for Windows, select the appropriate option from the menu.

Using the Flash downloader from ADW

Follow these steps to use the Flash downloader from ADW:

1. Select **Flash Download…** from the **File** menu. The Flash Download dialog is displayed (Figure 8-1).

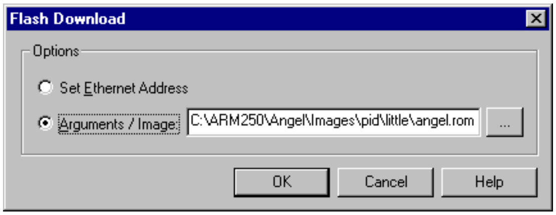


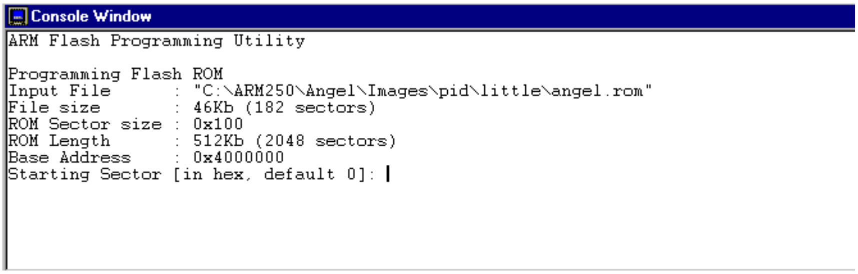
Figure 8-1 Flash Download dialog

2. Enter a pathname or click **Browse** to select a binary file to download.

Note

The pathname to the binary file must not contain spaces.

3. Click **OK**. The Flash downloader reads the binary file and request a start address in the console window of the Debugger (Figure 8-2).



```
ARM Flash Programming Utility

Programming Flash ROM
Input File      : "C:\ARM250\Angel\Images\pid\little\angel.rom"
File size       : 46Kb (182 sectors)
ROM Sector size : 0x100
ROM Length      : 512Kb (2048 sectors)
Base Address    : 0x4000000
Starting Sector [in hex, default 0]: |
```

Figure 8-2 Entering a start address

4. Enter an address. The console window displays the progress as the Flash is written.

Using the Flash downloader from armsd

To use the Flash downloader from the command line, write a batch file containing this command:

```
armsd -adp -port s,p -line 38400 -exec flash ROMname
```

where:

`flash` is the name of the flash downloader, either:

- `ARM250\Bin\flash.li` for a little-endian system
- `ARM250\Bin\flash.bi` for a big-endian system.

`ROMname` is the name of the binary file that you want to be programmed into Flash memory.

Note

The pathname to the binary file must not contain spaces.

Execute the batch file to download to Flash. Enter the address to start writing from when prompted to do so.

Refer to the *ARM Target Development System User Guide* for more information.

Chapter 9
ARM Procedure Call Standard

This chapter describes the ARM Procedure Call Standard. It contains the following sections:

- *About the ARM Procedure Call Standard* on page 9-2
- *APCS definition* on page 9-6
- *C language calling conventions* on page 9-16
- *Function entry examples* on page 9-18
- *Function exit* on page 9-24.

9.1 About the ARM Procedure Call Standard

The *ARM Procedure Call Standard* (APCS) is a set of rules that regulates and facilitates calls between separately compiled or assembled program fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions
- passing of machine-level arguments and the return of machine-level results at externally visible function or procedure calls.

Because ARM processors are used in a wide variety of systems, the APCS is not a single standard but a consistent family of standards. (See *APCS variants* below for details of the variants in the family.) When you implement systems such as runtime systems, operating systems, embedded control monitors, you must choose the variant or variants most appropriate to your requirements.

The different members of the APCS family are not binary compatible. If you are concerned with long-term binary compatibility you must choose your options carefully.

Throughout this chapter, the term *function* is used to mean function, procedure, or subroutine.

9.1.1 APCS variants

Previous versions of the toolkit supported 48 basic APCS variants, derived from 5 independent choices (2 x 2 x 3 x 2 x 2), as follows:

- Address size:
    - 32-bit
    - 26-bit.
- Reentrancy:
    - non-reentrant
    - reentrant.
- Floating-point architecture:
    - software floating-point (no floating-point hardware)
    - hardware floating-point with floating-point arguments passed in floating-point registers
    - hardware floating-point with floating-point arguments not passed in floating-point registers.

- Stack limit checking:
  - no software stack checking
  - software stack checking.
- Call frames linked through a frame pointer register:
  - No frame pointer register used. Call frames not linked.
  - Frame pointer register used. Call frames linked.
- Wide and narrow arguments:
  - Passing C/C++ language narrow arguments in widened form. In previous versions of the toolkit, this was the default:
  - Passing C/C++ language narrow arguments in narrow form. This is the default for this version of the toolkit.

For this release of the toolkit:

- Only 32-bit APCS variants are supported. 26-bit APCS variants are no longer supported, and are not documented.
- Only non-reentrant APCS variants are supported. Reentrant APCS variants are obsolete. These variants are documented for backwards compatibility only.
- For hardware floating-point architectures, floating-point arguments are passed in floating-point registers by default. APCS variants that pass floating-point arguments in integer registers are used only for interworking with the TPCS. Refer to Chapter 10 *Thumb Procedure Call Standard* for more information on the TPCS.
- Only APCS variants that do not require a frame pointer register are supported. APCS variants that require a frame pointer register are obsolete. These variants are documented for backwards compatibility only.
- Narrow arguments are passed in narrow form by default. However, `/wide` remains a fully supported option.
- The toolkit continues to support fully the options to build for ARM-Thumb interworking or not.

  For the ARM compilers and assembler, these options are specified with:

  ```
  toolname -apcs [/interwork|/nointerwork]
  ```

The following choices are supported for this release of the toolkit. Code conforming to one APCS variant is not compatible with code conforming to another:

Stack limit checking

The APCS defines conventions for software stack limit checking sufficient to support efficiently most requirements, including those of multiple threads and chunked stacks.

For the ARM compilers and assembler, these options are specified with:

```
toolname -apcs [/swst|/noswst]
```

Use `/swst` if you think that your stack might overflow and not be trapped by memory management hardware. Use `/noswst` to get the smallest, fastest code.

Floating-point architecture

The APCS variant you choose will depend on your floating-point hardware. If your system includes a hardware floating-point unit, use the hardware floating-point APCS variant. If not, use the software floating-point APCS variant. For the ARM compilers and assembler, these options are specified with:

```
toolname -fpu [fpa|none]
```

or:

```
toolname -apcs [/hardfp|/softfp]
```

Use the `-fpu` option in preference to `/hardfp` and `/softfp`.

If hardware floating-point is selected you can choose whether floating-point arguments are passed in floating-point registers:

Arguments in floating-point registers

This is the default.

Arguments not in floating-point registers

Use this option to interwork with the TPCS. Thumb cannot use hardfp.

For the ARM compilers and assembler, these options are specified with:

```
toolname -apcs /hardfp[/fpr|/nofpr]
```

Reentrant or non-reentrant

This option is obsolete. You should use reentrant variants of the APCS only if it is necessary to support existing code.

The reentrant variant of the APCS supports the generation of code that is free of relocation directives. This is position-independent code that addresses all data indirectly through a static base register.

Frame pointer or no frame pointer

Requiring the use of a frame pointer register is obsolete. You should use APCS variants that use a frame pointer register only if it is necessary to support existing code.

For the ARM compilers and assembler, these options are specified with:

```
toolname -apcs [/fp|/nofp]
```

9.2 APCS definition

This section defines the ARM Procedure Call Standard. Where examples are given, these do not form part of the standard. They are given only to aid clarity.

9.2.1 APCS conformance

The APCS defines two levels of conformance:

conforming A program fragment that conforms to the APCS while making a call to an external function (one that is visible between compilation units) is said to be *conforming*.

strictly conforming

A program that conforms to the APCS at all instants of execution is said to be *strictly conforming*.

In general, compiled code is expected to be strictly conforming. Hand written assembly language code is expected to be conforming.

Whether or not program fragments for a particular ARM-based environment are required to conform strictly to the APCS is part of the definition of that environment.

9.2.2 APCS register names and roles

The ARM architecture with the FPA coprocessor defines:

- 15 visible general purpose integer registers
- a 32-bit program counter register
- eight floating-point registers.

In non-user processor modes, some general purpose registers are banked. In all modes, the availability of the floating-point instruction set depends on the processor model, hardware, and operating system. Refer to the *ARM Architectural Reference Manual* for a detailed description of the ARM register banks and processor modes.

Table 9-1, below, and Table 9-2 on page 9-8 give the names and functions of the ARM registers under the APCS.

Table 9-1 APCS registers

Register	APCS name	APCS role
r0	a1	argument 1/scratch register/result
r1	a2	argument 2/scratch register/result
r2	a3	argument 3/scratch register/result
r3	a4	argument 4/scratch register/result
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	sb/v6	static base/register variable
r10	sl/v7	stack limit/stack chunk handle/register variable
r11	fp/v8	frame pointer/register variable
r12	ip	scratch register/new-sb in inter-link-unit calls
r13	sp	lower end of the current stack frame
r14	lr	link register/scratch register
r15	pc	program counter

General purpose registers

The 16 integer registers are divided into 3 sets:

- four argument registers that can also be used as scratch registers or as caller-saved register variables
- five callee-saved registers, conventionally used as register variables
- seven registers that have a dedicated role, at least some of the time, in at least one variant of the APCS (see *APCS variants* on page 9-2).

The registers sp, lr and pc have dedicated roles in all variants of the APCS.

The ip register has a dedicated role only during function call. At other times it may be used as a scratch register. Conventionally, ip is used by compiler code generators as a local code generator temporary register.

There are dedicated roles for sb, fp, and sl in some variants of the APCS. In other variants they may be used as callee-saved registers.

The APCS permits lr to be used as a register variable when it is not in use during a function call. It further permits an ARM system specification to forbid such use in some, or all, non-user ARM processor modes.

Floating-point registers (FPA Architecture)

Each ARM floating-point (FP) register holds one FP value of single, double, extended or internal precision. A single precision value occupies one machine word. A double precision value occupies two words. An extended precision value occupies three words, as does an internal precision value.

Floating-point registers are divided into two sets, analogous to the subsets a1 through a4 and v1 through v5/v7 of the general registers:

- Registers f0 through f3 need not be preserved by called functions. f0 is the FP result register, and f0 through f3 may hold the first four FP arguments. See *Data representation and argument passing* on page 9-14 and *APCS variants* on page 9-2.
- Registers f4 through f7, the *variable* registers, preserved by callees.

Table 9-2 APCS floating-point registers

Name	Number	APCS Role
f0	0	FP argument 1/FP result/FP scratch register
f1	1	FP argument 2/FP scratch register
f2	2	FP argument 3/FP scratch register
f3	3	FP argument 4/FP scratch register
f4	4	floating-point register variable
f5	5	floating-point register variable
f6	6	floating-point register variable
f7	7	floating-point register variable

9.2.3 The stack

The APCS supports both contiguous and chunked stacks. The stack must be readable and writable by the executing program.

Chunked stacks

Chunked stacks have the following properties:

- A chunked stack can be extended by allocating additional chunks anywhere in memory.
- A chunked stack is a singly-linked list of *activation records*, linked through a *stack backtrace data structure* (see *The stack backtrace data structure* on page 9-11), stored at the high-address end of each activation record.
- Each contiguous chunk of the stack must be allocated to activation records in descending address order. At all instants of execution, sp must point to the lowest used address of the most recently allocated activation record.
- There are no constraints on the ordering of multiple stack chunks in the address space.
- A chunked stack must be limit-checked. It can be extended only when a limit check fails.

Contiguous stacks

Contiguous stacks have the following properties:

- A contiguous stack can be extended at its low address only.
- The activation records in a contiguous stack need not be linked together through a stack backtrace data structure.

Stack chunk limit

Associated with sp is a possibly implicit stack chunk limit, below which sp must not be decremented.

At all instants of execution, the memory between sp and the stack chunk limit must contain nothing of value to the executing program. It may be modified unpredictably by the execution environment.

There are two types of stack chunk limit:

implicit The stack chunk limit is said to be *implicit* if chunk overflow is detected and handled by the execution environment. If the stack chunk limit is implicit, sl may be used as v7, an additional callee-saved variable register.

explicit The stack chunk limit is said to be *explicit* if chunk overflow is detected and handled by the program and its library support code.

If the conditions of the remainder of this subsection hold at all instants of execution, the program conforms strictly to the APCS. If they hold at, and during, external function calls (visible between compilation units), the program conforms to the APCS.

If the stack chunk limit is explicit, sl must:

- point at least 256 bytes above it
- identify the current stack chunk in a system-defined manner
- at all times, identify the same chunk as sp points into.

The values of sl, fp and sp must be multiples of 4.

(`sl >= stack_chunk_limit + 256` allows the most common limit checks to be made very cheaply during function entry.)

This final requirement implies that on changing stack chunks, registers sl and sp must be loaded simultaneously using:

```
LDM ..., {..., sl, sp}.
```

In general, this means that return from a function executing on an extension chunk to one executing on an earlier-allocated chunk should be through an intermediate function invocation, specially fabricated when the stack was extended.

9.2.4 The stack backtrace data structure

For chunked stacks, the value in fp must be zero or must point to a list of stack backtrace data structures that describe the sequence of outstanding function calls. If this constraint holds when external functions are called, the program is conforming. If it holds at all instants of execution, the program is strictly conforming.

The stack backtrace data structure shows between four and 27 words, with those words higher on the page being at higher addresses in memory. The values shown in brackets are optional, and their presence need not imply the presence of any other. The floating-point values are stored in an internal format, and occupy three words each. Figure 9-1 on page 9-11 shows the stack backtrace structure.

```
save code pointer          [fp, #0]      <-fp points to here
return link value          [fp, #-4]
return sp value            [fp, #-8]
return fp value            [fp, #-12]
{saved v7 value}
{saved v6 value}
{saved v5 value}
{saved v4 value}
{saved v3 value}
{saved v2 value}
{saved v1 value}
{saved a4 value}
{saved a3 value}
{saved a2 value}
{saved a1 value}
{saved f7 value}           three words
{saved f6 value}           three words
{saved f5 value}           three words
{saved f4 value}           three words
```

Figure 9-1 Stack backtrace structure

The save code pointer

The save code pointer (the value of pc) is the value of the pc at the instruction that starts to create the stack backtrace structure. Typically this is a store multiple instruction of the form:

```
    STMFD   sp!, {optional_registers,old_fp,sp,lr,pc}
```

The save code pointer (the value of pc) allows the function corresponding to a stack backtrace structure to be located.

Function exit

Table 9-3 shows the registers to which the values in the stack backtrace structure are restored on function exit.

Table 9-3 Function exit

Value	Restored to
return link value	pc
return sp value	sp
return fp value	fp

9.2.5 Function invocations and stack backtrace structures

Note

The following applies only to APCS variants that use a frame pointer register.

If function invocation A calls function B, then A is termed a *direct ancestor* of the invocation of B. If invocation A[1] calls invocation A[2] calls... calls B, then each of the A[i] is an ancestor of B and invocation A[i] is *more recent* than invocation A[j] if i > j.

The return fp value must be 0, or must be a pointer to a stack backtrace data structure created by an ancestor of the function invocation that created the backtrace structure pointed to by fp. No more recent ancestor must have created a backtrace structure. (There may be any number of tail called invocations between invocations that create backtrace structures.)

9.2.6 Control arrival

At the instant when control arrives at the target function:

- pc contains the address of an entry point to the target function (reentrant functions may have two entry points).
- lr contains the value to restore to pc on exit from the function (the return link value. See *The stack backtrace data structure* on page 9-11).
- sp points at or above the current stack chunk limit. If the limit is explicit, it must point at least 256 bytes above it. See *The stack* on page 9-9.

- For APCS variants that use a frame pointer register, fp contains 0 or points to the most recently created stack backtrace structure. See *The stack backtrace data structure* on page 9-11.
- The space between sp and the stack chunk limit is readable and writable memory that the called function can use as temporary workspace and overwrite with any values before the function returns. See *The stack* on page 9-9.
- Arguments are marshalled as described in *Data representation and argument passing* on page 9-14.

Reentrant functions

Note

This section describes obsolete facilities. It is provided for backwards compatibility only.

A reentrant target function has two entry points. Control arrives:

- at the *intra-link-unit entry point* if the caller has been directly linked with the callee
- at the *inter-link-unit entry point* if the caller has been separately linked with a *stub* of the callee.

In non-static-data-using leaf functions, the two entry points are at the same address. Otherwise they are separated by a single instruction.

On arrival at the intra-link-unit entry point, sb must identify the static data of the link unit that contains both the caller and the callee.

On arrival at the inter-link-unit entry point, either:

- ip must identify the static data of the link unit containing the target function.
- The target function must make neither direct nor indirect use of static data. In practice this usually means that the callee must be a leaf function that makes no direct use of static data.

The way in which sb *identifies* the static data of a link unit is not specified by the APCS.

If the call is by tail continuation, *calling function* means the function that will be returned to if the tail continuation is converted to a return.

If code is not required to be reentrant or shareable, sb may be used as v6, an additional variable register. (See Table 9-1 on page 9-7.)

9.2.7 Data representation and argument passing

Argument passing in the APCS is defined in terms of an ordered list of machine-level values passed from the caller to the callee, and a single word or floating-point result passed back from the callee to the caller. Each value in the argument list is either:

- an integer value of size one word
- a floating-point value of size one, two, or three words.

A callee may corrupt any of its arguments, however passed.

The APCS does not define:

- the layout in store of records, arrays, as so forth, used by ARM-targeted compilers for C and C++
- the order in which language-level arguments are mapped into their machine-level representations.

This means that, the mapping from language-level data types and arguments to APCS words is defined by each language implementation, not by the APCS. There is no reason why two ARM-targeted implementations of the same language cannot use different mappings and not support cross-calling.

Implementors are encouraged to adopt not just the APCS standard, but to accommodate the natural mappings of source language objects into argument words. Guidance about this is given in *C language calling conventions* on page 9-16.

At the instant control arrives at the target function, the argument list must be allocated as follows:

- in the APCS variants that support the passing of floating-point arguments in floating-point registers (see *APCS variants* on page 9-2), the first four floating-point arguments (or fewer if the number of floating-point arguments is less than four) are in machine registers f0 through f3. Refer to Chapter 6 *Using the Procedure Call Standards* in the *ARM Software Development Toolkit User Guide* for more information.
- the first four remaining argument words (or fewer if there are fewer than four argument words remaining in the argument list) are in machine registers a1 through a4
- the remainder of the argument list (if any) is in memory, at the location addressed by sp and higher addressed words from this point on.

A floating-point value not passed in a floating-point register is treated as one, two, or three integer values, according to its precision.

9.2.8 Control return

When the return link value for a function call is placed in the pc:

- sp, fp, sl/v7, sb/v6, v1-v5, and f4-f7 must contain the same values as they did at the instant of control arrival.
- If the function returns a simple value of one word or less, the value must be in a1. A language implementation is not obliged to consider *all* single-word values simple. See *Non-simple value return* on page 9-17 for more information.
- If the function returns a simple floating-point value, the value must be in f0 for hardfp APCS variants. For softfp variants, a floating-point result is returned in r0, or r0 and r1.

The values of ip, lr, a2-a4, f1-f3 and any stacked arguments are undefined. The definition of control return means that this is a *callee saved* standard. The caller's CPSR flags are not preserved across a function call.

9.3 C language calling conventions

This section describes the conventions that apply to C language arguments and return values.

9.3.1 Argument representation

A floating-point value occupies one, two, or three words, as appropriate to its type. Floating-point values are encoded in IEEE 754 format, with the most significant word of a **`double`** having the lowest address. Refer to Chapter 11 *Floating-point Support* for more information on floating-point support.

`char`, **`short`**, **`pointer`** and other integral values occupy one word in an argument list. **`long long`** (__int64) values occupy two words.

On the ARM, characters are naturally unsigned. ANSI C and C++ have signed, unsigned, and plain (do not care) **`char`** types. Classic C does not have the signed **`char`** type, so plain **`char`** must be considered signed. In PCC mode (`-pcc` compiler option), the C compilers treat a plain **`char`** as signed, and widen its value appropriately when it is used as an argument.

A structured value occupies an integral number of integer words, even when it contains only floating-point values.

9.3.2 Argument list marshalling

Argument values are marshalled in the order written in the source program.

If the called function accepts a fixed number of arguments, and if passing floating-point arguments in FP registers, the first four floating-point arguments are loaded into FP registers.

The first four of the remaining argument words are loaded into a1 through a4, and the remainder are pushed onto the stack in reverse order. As a consequence, an FP value can be passed in integer registers, or even split between an integer register and the stack.

Arguments later in a stacked argument list have higher addresses than those earlier in the stacked argument list.

9.3.3 Non-simple value return

A non-simple type is any non floating-point type greater than one word in size (including structures containing only floating-point fields), and certain one-word structured types.

A structure is termed *integer-like* if its size is less than or equal to one word, and the offset of each of its addressable subfields is zero. An integer-like structured result is considered simple and is returned in a1.

The following are both integer-like:

```
struct {int a:8, b:8, c:8, d:8;}
union {int i; char *p;}
```

The following is not integer-like:

```
struct {char a; char b; char c; char d;}
```

A multi-word or non integer-like result is returned to an address passed as an additional first argument to the function call.

At machine level:

```
TT tt = f(x, ...);
```

is implemented as:

```
TT tt; f(&tt, x, ...);
```

9.4 Function entry examples

A complete discussion of function entry is complex. This section covers some of the most important issues and special cases.

Note

This section includes descriptions of the reentrant variant of the APCS, and the format of the stack backtrace structure. This information is obsolete and is provided for backwards compatibility only.

The important issues for function entry are:

- establishing the static base (obsolete reentrant APCS variants only)
- creating the stack backtrace data structure (only for APCS variants that use a frame pointer register)
- saving the floating-point variable registers if required
- checking for stack overflow if the stack chunk limit is explicit.

9.4.1 Definitions

The following terms are used to identify particular types of functions.

Leaf functions

A function is termed *leaf* if its body contains no function calls. A leaf function that makes no use of static data need not establish a static base.

Tail calls or tail continuation functions

If function F calls function G immediately before an exit from F, the call-exit sequence can often be replaced instead by a *return to G*. After this transformation, the return to G is called a *tail call* or *tail continuation*.

There are many subtle considerations when using tail continuations. If stacked arguments are unstacked by callers (almost mandatory for variadic callees), G cannot be directly tail called if G itself takes stacked arguments. This is because there is no return to F to unstack them.

If this call to G takes fewer arguments than the current call to F, some of F's stacked arguments can be replaced by G's stacked arguments. However, this may not be easy to assert if F is variadic. There may be no tail call of G if the address of any of F's arguments or local variables has *leaked out* of F. This is because on return to G, the address may be invalidated by adjustment of the stack pointer. In general, this precludes tail calls if any local variable or argument has its address taken.

Frameless functions

For APCS variants that use a frame pointer register, a function does not need to create a stack backtrace structure if it uses no v-registers and either:

- it is a leaf function
- all the function calls it makes from its body are tail calls.

Such functions are termed *frameless.*

Stack backtrace structures need not be created for APCS variants that do not use a frame pointer register.

9.4.2 Establishing the static base

Note

Reentrant variants of the APCS are obsolete.

A reentrant function can be entered directly via a call from the same link unit (an *intra-link-unit call*), or indirectly through a function pointer or direct call from another link unit (an *inter-link-unit call*).

The general scheme for establishing the static base in reentrant code is:

```
intra   MOV ip, sb  ; intra link unit (LU) calls target here
inter               ; inter-LU calls target here, having loaded
                    ; ip via an inter-LU or fn-pointer veneer.

    ; create backtrace structure, saving sb

        MOV sb, ip  ; establish sb for this LU

    ; rest of entry
```

Code that does not have to be reentrant does not need to use a static base. Code that is reentrant is marked as such, allowing the linker to create the inter-link-unit veneers needed between independent reentrant link units, and between reentrant and non-reentrant code.

9.4.3 Creating the stack backtrace structure

For non-reentrant, nonvariadic functions, a stack backtrace structure can be created using three instructions:

```
MOV     ip, sp              ; save current sp,
                            ; ready to save as old sp

STMFD   sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc}
                            ;as needed
SUB     fp, ip, #4
```

Each argument register a1 through a4 has to be saved only if a memory location is needed for the corresponding parameter, either because it has been spilled by the register allocator or because its address has been taken.

Each of the registers v1 through v7 has to be saved only if used by the called function. The minimum set of registers to be saved is {fp, old-sp, lr, pc}.

A reentrant function must avoid using ip in its entry sequence:

```
STMFD  sp!, {sp, lr, pc}
STMFD  sp!, {a1-a4, v1-v5, sb, fp}      ; as needed
ADD    fp, sp, #8+4*|{a1-a4, v1-v5, sb, fp}|
                                        ; as used above
```

sb (also known as v6) must be saved by a reentrant function if it calls any function from another link unit (which would alter the value in sb). This means that, in general, sb must be saved on entry to all non-leaf, reentrant functions.

For variadic functions the entry sequence is still more complicated. Usually, you have to make a contiguous argument list on the stack. For non-reentrant variadic functions, use:

```
MOV     ip, sp          ; save current sp, ready to
                        ; save as old sp
STMFD   sp!,{a1-a4}     ; push arguments on stack
STMFD   sp!,{v1-v6, fp, ip, lr, pc}
                        ; push other registers on
                        ; stack as needed
SUB     fp, ip, #20     ; if all of a1-a4 pushed...
```

It is not necessary to push arguments corresponding to fixed parameters, though saving a1-a4 is little more expensive than just saving, say, a3-a4.

Floating-point arguments are never passed to variadic functions in floating-point registers.

9.4.4 Saving and restoring floating-point registers

The Issue 2 FPA instruction set defines two instructions for saving and restoring the floating-point registers:

- Store Floating Multiple (`SFM`)
- Load Floating Multiple (`LFM`).

These are as follows:

- `SFM` and `LFM` are exact inverses
- `SFM` will never trap, whatever the IEEE trap mode and the value transferred (unlike `STFE` which can trap on storing a signalling NaN)
- `SFM` and `LFM` transfer 3-word internal representations of floating-point values which vary from implementation to implementation, and which, in general, are unrelated to any of the supported IEEE representations
- any 1-4, cyclically contiguous floating-point registers can be transferred by `SFM`/`LFM` (for example, {f4-f7}, {f6, f7, f0}, {f7, f0}, {f1}).

Function entry

On function entry, a typical use of `SFM` might be as follows:

```
SFMFD  f4, 4, [sp]!     ; save f4-f7 on a
                        ; Full Descending stack,
                        ; adjusting sp as values are pushed.
```

Function exit

On function exit, the corresponding sequence might be:

```
LFMEA  f4, 4, [fp, #-N] ; restore f4-f7
                        ; fp-N points just
                        ; above the floating-point
                        ; save area.
```

For chunked stacks, sp-relative addressing may be unavailable on function exit if the stack has been discontiguously extended.

9.4.5 Checking for stack limit violations

In some environments, stack overflow detection is implicit. An off-stack reference causes an address error or memory fault, which may in turn cause stack extension or program termination.

In other environments, the validity of the stack must be checked on function entry, and at other times if the function:

- Uses 256 bytes or less of stack space.
- Uses more than 256 bytes of stack space, but the amount is known and bounded at compile time.
- Uses an amount of stack space unknown until runtime. This does not arise in C, apart from in stack-based implementations of the non-standard, BSD-UNIX `alloca()` function. The APCS does not easily support `alloca()`.

The check for stack limit violation is made at the end of the function entry sequence, by which time ip is available as a work register.

If the check fails, a standard runtime support function is called, either `__rt_stkovf_split_small` or `__rt_stkovf_split_big`.

Any environment that supports explicit stack-limit checking must provide functions that can do one of the following:

- terminate execution
- extend the existing stack chunk, and decrement sl
- allocate a new stack chunk, reset sp and sl to point into it, and guarantee that an immediate repeat of the limit check will succeed.

Stack limit checking (small, fixed frames)

For frames of 256 bytes or less the limit check is as follows:

```
; create the activation record.
    CMPS    sp, sl
    BLLO    |__rt_stkovf_split_small|
    SUB     sp, sp, #size of locals     ; <= 256, by hypothesis
```

This adds two instructions and, in general, only two cycles to function entry.

After a call to `__rt_stkovf_split_small`, fp and sp do not necessarily point into the same stack chunk. Arguments passed on the stack must be addressed by offsets from fp, not by offsets from sp.

Stack limit checking (large, fixed frames)

For frames bigger than 256 bytes, the limit check proceeds as follows:

```
    SUB     ip, sp, #FrameSizeBound ; can do in 1 instr
    CMPS    ip, sl
    BLLO    |__rt_stkovf_split_big|
    SUB     sp, sp, #InitFrameSize  ; may require > 1 instr
```

where:

`FrameSizeBound`

can be any convenient constant at least as big as the largest frame the function will use.

`InitFrameSize`

is the initial stack frame size. Subsequent adjustments within the called function require no limit check.

Note

Functions containing nested blocks may use different amounts of stack at different instants during their execution.

After a call to `__rt_stkovf_split_big`, fp and sp do not necessarily point into the same stack chunk. Arguments passed on the stack must be addressed by offsets from fp, not by offsets from sp.

9.5 Function exit

Function exit can usually be implemented in a single instruction (this is not the case if floating-point registers have to be restored). Typically, there are at least as many function exits as entries, so it is always advantageous to move an instruction from an exit sequence to an entry sequence.

If exit is a single instruction, further instructions can be saved in multi-exit functions by replacing branches to a single exit with the exit instructions themselves.

Saving and restoring floating-point registers is discussed in *Saving and restoring floating-point registers* on page 9-21.

To exit from functions that use no stack and save no floating-point registers, use:

```
MOV    pc, lr
```

or:

```
BX  lr
```

for interworking code.

To exit from other functions that use a frame pointer register and save no floating-point registers, use a pre-decrement load multiple (`LDMEA`):

```
LDMEA  fp, {v1-v5, sb, fp, sp, pc}  ; as saved
```

Here, fp must point just below the save code pointer, as this value is not restored.

A function that does not use a frame pointer register must unwind the stack in its exit sequence. Generally this means:

- Increment sp by the amount that it was explicitly decremented in the function body.
- Pop saved registers from the stack in the reverse order to that in which they were pushed.

Chapter 10
Thumb Procedure Call Standard

This chapter describes the Thumb procedure call standard. It contains the following sections:

10.1 About the Thumb Procedure Call Standard

The *Thumb Procedure Call Standard* (*TPCS*) is a set of rules that govern inter-calling between functions written to the Thumb subset of the ARM instruction set.

The TPCS is a cut-down version of the APCS. If you are unfamiliar with the APCS and its terminology, you will find it helpful to read Chapter 9 *ARM Procedure Call Standard* before continuing with this chapter.

Specifically, the TPCS does not allow:

- Disjoint stack extension (stack chunks). Under the TPCS, the stack must be contiguous. However, this does not necessarily prohibit the use of co-routines.
- Calling the same entry point with different sets of static data (multiple instantiation, or reentrancy). Multiple instantiation can be implemented at a user level, by placing in a **`struct`** all variables that need to be multiply instantiated, and passing each function a pointer to the **`struct`**.
- Hardware floating-point. Thumb code cannot access floating-point (FP) instructions without switching to ARM state. Floating-point is supported indirectly by defining how FP values are passed to, and returned from, Thumb functions in integer registers.

10.2 TPCS definition

This section defines the Thumb Procedure Call Standard. Where examples are given these do not form part of the standard. They are given only to aid clarity.

10.2.1 TPCS register names

The Thumb register subset has:

- eight visible general purpose registers (r0-r7), referred to as the *low* registers
- a stack pointer (sp)
- a link register (lr)
- a program counter (pc).

In addition, the Thumb subset can access the rest of the ARM registers (r8-r15, referred to as the *high* registers) singly through a set of special instructions. See the *ARM Architectural Reference Manual* for details.

Table 10-1 on page 10-4 shows the names and roles of the Thumb registers in the context of the TPCS.

Table 10-1 TPCS registers

Register	TPCS name	TPCS role
r0	a1	argument 1/scratch register/result
r1	a2	argument 2/scratch register/result
r2	a3	argument 3/scratch register/result
r3	a4	argument 4/scratch register/result
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4/wr	register variable/work register in function entry/exit
r8	(v5)	(ARM v5 register, no defined role in Thumb)
r9	(v6)	(ARM v6 register, no defined role in Thumb)
r10	sl (v7)	stack limit
r11	fp (v8)	frame pointer (usually not used in Thumb state)
r12	(ip)	(ARM ip register, no defined role in Thumb. May be used as a temporary register on Thumb function entry/exit.)
r13	sp	stack pointer (full descending stack)
r14	lr	link register
r15	pc	program counter

10.2.2 The Stack

The stack contains a series of activation records allocated in descending address order. These activation records may be linked through a stack backtrace data structure but there is no obligation for code under the TPCS to create a stack backtrace structure.

A stack limit is said to be *implicit* if stack overflow is detected and handled by the execution environment, otherwise it is *explicit*. Associated with sp is a possible *implicit stack limit*, below which sp must not be decremented unless a suitable trapping mechanism is in place to detect below-limit reads or writes.

At all instants of execution, the memory between sp and the stack limit must contain nothing of value to the executing program. This memory may be modified unpredictably by the execution environment.

If the stack limit is explicit, sl must point at least 256 bytes above it. The values of sl, fp and sp are multiples of 4.

Note

fp is usually not used in Thumb state.

Implicit or explicit stack limit checking

Stack limit checking may be:

implicit performed by the memory management hardware

explicit performed by the program and its library support code.

The TPCS defines conventions for software stack-limit checking sufficient to support most requirements.

10.2.3 Control arrival

At the instant when control arrives at the target function:

- pc contains the address of an entry point to the target function.
- lr contains the value to restore to pc on exit from the function (the return link value, see *The stack backtrace data structure* on page 9-11).
- sp points at or above the current stack limit. If the limit is explicit, sp will point at least 256 bytes above it (see *The Stack* on page 10-4).
- If the function is built to use a frame pointer register, fp contains 0 or points to the most recently created stack backtrace structure (see *The stack backtrace data structure* on page 9-11). This is not usual in Thumb state.
- The space between sp and the stack limit must be readable and writable memory which the called function can use as temporary workspace, and overwrite with any values before the function returns (see *The Stack* on page 10-4).

Arguments are marshalled as described below.

10.2.4 Data representation and argument passing

Argument passing in the TPCS is defined in terms of an ordered list of machine-level values passed from the caller to the callee, and a single-word or floating-point result passed back from the callee to the caller. Each value in the argument list must be either:

- an integer value of size one word
- a floating-point value of size one, two, or three words.

A callee may corrupt any of its arguments, however passed.

At the instant control arrives at the target function, the argument list is allocated as follows:

- the first four argument words (or fewer if there are fewer than four argument words remaining in the argument list) are in machine registers a1-a4
- the remainder of the argument list (if any) is in memory, at the location addressed by sp and higher addressed words thereafter.

A floating-point value is treated as one, two, or three integer values, as appropriate to its precision. The TPCS does not support the passing or returning of floating-point values in ARM floating-point registers.

10.2.5 Control return

When the return link value for a function call is placed in the pc:

- sp, fp, sl, v6, v5, and v1-v4 contain the same values as they did at the instant of control arrival. If the function returns a simple value of size one word or less, the value is contained in a1.
- If the function returns a simple value of size one word or less, then the value must be in a1. A language implementation is not obliged to consider *all* single-word values simple. See *Non-simple value return* on page 10-8.
- If the function returns a simple floating-point value, the value is encoded in a1, {a1, a2}, or {a1, a2, a3}, depending on its precision.

Note

fp is usually not used by Thumb state code.

10.3 C language calling conventions

This section describes the conventions that apply to C language arguments and return values.

10.3.1 Argument representation

A floating-point value occupies one, two, or three words, as appropriate to its type. Floating-point values are encoded in IEEE 754 format, with the most significant word of a double having the lowest address.

`char`, **`short`**, **`pointer`** and other integral values occupy one word in an argument list. **`long long`** (__int64) values occupy two words in an argument list. Structure values are treated as a sequence of integer words, even if all fields have floating-point type.

Characters are naturally unsigned. ANSI C has signed, unsigned, and plain chars.

10.3.2 Argument list marshalling

Argument values are marshalled in the order written in the source program.

The first 4 argument words are loaded into a1-a4, and the remainder are pushed onto the stack in reverse order. This means that arguments later in the argument list have higher addresses than those earlier in the argument list. As a consequence, a floating-point value can be passed in integer registers, or even split between an integer register and the stack.

10.3.3 Non-simple value return

A non-simple type is any non-floating-point type of size greater than one word (including structures containing only floating-point fields), and certain single-word structured types.

A structure is considered integer-like if its size is less than or equal to one word, and the offset of each of its addressable subfields is zero. An integer-like structured result is considered simple and is returned in register a1.

Integer-like structures:

```
struct {int a:8, b:8, c:8, d:8;} union {int i; char *p;}
```

Non integer-like structures:

```
struct {char a; char b; char c; char d;}
```

A multi-word or non-integer-like result is returned to an address passed as an additional first argument to the function call. At the machine level:

```
TT tt = f(x, ...);
```

is implemented as:

```
TT tt; f(&tt, x, ...);
```

10.4 Function entry examples

A complete discussion of function entry is complex. This section discusses a few of the most important issues and special cases.

10.4.1 Definitions

The following terms are used to identify particular types of functions.

Tail calls or tail continuation functions

If function F calls function G immediately before an exit from F, the call-exit sequence can often be replaced instead by a return to G. After this transformation, the return to G is called a *tail call* or *tail continuation*.

Note

Tail continuation is difficult with the Thumb instruction set because of the limited range of the B instruction (+/-2048 bytes).

10.4.2 Simple function entry

The simplest entry sequence for functions is:

```
PUSH{save-registers, lr} ; Save registers as needed.
```

The corresponding exit sequence is:

```
POP {save-registers, pc}
```

It is sometimes necessary to save {a1-a4} before {v1-v4}, if the arguments can be addressed as a single array of arguments accessed from the address of one of the saved argument registers.

In this case, the function entry sequence becomes:

```
PUSH{a1-a4}             ; as necessary
PUSH{save-registers, lr}
```

and the corresponding exit sequence becomes:

```
POP {save-registers}
POP {a3}
ADD sp, sp, #16
MOV pc, a3
```

10.4.3 Checking for stack limit violations

In some environments, stack overflow detection is implicit. An off-stack reference causes an address error or memory fault which may, in turn, cause stack extension or program termination.

In other environments, the availability of stack space must be checked whenever sp is decremented (at least on function entry).

The check for stack limit violation is made at the end of the function entry sequence. If the check fails, one of the following standard runtime support functions is called:

- `__16__rt_stkovf_split_small`
- `__16__rt_stkovf_split_big`.

Each environment that supports explicit stack limit checking must provide these functions to either:

- terminate execution
- extend the existing stack, decrementing sl.

Stack limit checking: small, fixed frames

For frames of 256 bytes or less, the limit check may be implemented as follows:

```
    CMP sp, sl
    BHS no_ovf
    BL  |__16__rt_stkovf_split_small|
no_ovf
```

Stack limit checking: large, fixed frames

For frames larger than 256 bytes, the limit check may be implemented as follows:

```
    LDR wr, framesize
    ADD wr, sp
    CMP wr, sl
    BHS no_ovf
    BL  |__16__rt_stkovf_split_big|
no_ovf
    MOV sp,wr
    ; ...
    ALIGN
framesize
    DCD -Framesize
```

Note

Functions containing nested blocks may use different amounts of stack at different times during their execution. If this is the case, subsequent stack adjustments require no limit check if the initial stack check examines the maximum stack depth.

10.5 Function exit

To exit from a function that saves no registers use:

```
MOV pc, lr
```

or:

```
BX  lr
```

for interworking code.

where lr has the same value as it had on entry to the function.

To exit otherwise use:

```
POP {saved-registers, pc}
```

lr does not need to be preserved.

To exit from functions that create a stack backtrace structure, use:

```
LDR wr, [sp, #fp-offset]    ; Restore fp
MOV fp, wr
LDR a4, [sp, #lr_offset]    ; Get lr in a4
POP {saved-regs}
ADD sp, sp, #16+pushed-args*4; pushed-args*4 only
                            ; needed if variadic
MOV pc, a4                  ; Return
```

Chapter 11
Floating-point Support

This chapter describes the different types of floating-point mechanisms. It contains the following sections:

11.1 About floating-point support

Floating-point arithmetic on the ARM can be done in three ways:

- software floating-point library (fplib), supplied as part of the ARM C libraries
- hardware coprocessor (FPA) that executes a floating-point instruction set
- software floating point emulation (FPE), such as that used by the ARMulator.

When the floating-point library is used, the compiler makes calls to the library routines to do floating-point calculations. For the other options, the compiler uses floating-point instructions that are executed by either the FPA or the floating-point support code (FPASC), or emulated by a floating-point emulator.

The software floating-point library cannot use a hardware FPA, and it does not support some little-used facilities of the IEEE 754-1985 floating-point arithmetic standard. If either of these is required for your system, you are recommended to use FPASC.

Because use of floating-point library calls and use of floating-point instructions implies different procedure calling conventions, it is not possible to combine the two methods.

The ARM compilers default to the software floating-point library. You can switch armcc and armcpp to generate floating-point coprocessor instructions (see Chapter 6 Using the Procedure Call Standards in the *ARM Software Development Toolkit User Guide* for more information).

For more information on floating-point support, refer to:

- the ARM FPA 10 datasheet (ARM DDI 0020I)
- the IEEE standard for binary floating-point arithmetic (IEEE 754-1985).

11.1.1 Thumb

The Thumb C compiler does not generate floating-point instructions, because these are not available in the Thumb instruction set. The software floating-point library is the only option available for tcc.

11.2 The ARM floating-point library

The ARM software floating-point library provides a set of functions such as `_dadd` to add two doubles.

The code compiled using the ARM floating-point library cannot use the following facilities of the IEEE standard:

- underflow exceptions
- inexact exceptions
- rounding modes other than round to nearest
- extended precision.

11.2.1 Usage

The following APCS options control which floating-point mechanism is used by the compilers:

`/softfp` Use software floating-point library functions. This is the default for ARM processors without an FPU, and the only floating-point option available to Thumb compilers.

`/hardfp` Generate ARM coprocessor instructions for the FPA floating-point unit. You may also specify `/fpregargs` or `/nofpregargs`. The `/hardfp` and `/softfp` options are mutually exclusive. This option is not available for Thumb compilers.

Note that `/hardfp` implies `-fpu fpa`. Use the `-fpu` option in preference to `/softfp` and `/hardfp`.

Refer to Chapter 2 *The ARM Compilers* for more information.

11.2.2 Combining hardfp and softfp systems

With the software floating-point library, functions pass floating-point types in integer registers. FPA systems pass floating-point results in floating-point registers. The two return methods are not compatible. You should not mix ARM floating-point instructions and calls to the softfp library.

11.2.3 Floating-point library register usage

The software floating-point library provides a number of functions for basic floating-point operation. IEEE double precision (`double`) values are passed in pairs of registers, and single precision (`float`) numbers are passed in a single register.

For example, `_dadd` is the function to add two double precision numbers. It can be considered as having the prototype:

```
extern double _dadd(double, double);
```

That is, the two numbers to be added are passed in r0/r1 and r2/r3. The result is returned in r0/r1.

Similarly, the function `_fadd` (single precision add) has the two arguments passed in r0 and r1, and the result returned in r0. Table 11-1 gives the complete set of functions provided by the software floating-point library.

Table 11-1 Floating point library functions

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
_dadd	A+B	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_dsub	A–B	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_drsb	B–A	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_dmul	A*B	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_ddiv	A/B	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_drdv	B/A	r0/r1 (double)	r2/r3 (double)	r0/r1 (double)
_dneg	–A	r0/r1 (double)		r0/r1 (double)
_fadd	A+B	r0 (float)	r1 (float)	r0 (float)
_fsub	A–B	r0 (float)	r1 (float)	r0 (float)
_frsb	B–A	r0 (float)	r1 (float)	r0 (float)
_fmul	A*B	r0 (float)	r1 (float)	r0 (float)
_fdiv	A/B	r0 (float)	r1 (float)	r0 (float)
_frdv	B/A	r0 (float)	r1 (float)	r0 (float)
_fneg	–A	r0 (float)		r0 (float)
_dgr	A>B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)

Table 11-1 Floating point library functions (Continued)

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
_dgeq	A>=B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)
_dls	A<B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)
_dleq	A<=B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)
_dneq	A!=B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)
_deq	A==B	r0/r1 (double)	r2/r3 (double)	r0 (boolean)
_fgr	A>B	r0 (float)	r1 (float)	r0 (boolean)
_fgeq	A>=B	r0 (float)	r1 (float)	r0 (boolean)
_fls	A<B	r0 (float)	r1 (float)	r0 (boolean)
_fleq	A<=B	r0 (float)	r1 (float)	r0 (boolean)
_fneq	A!=B	r0 (float)	r1 (float)	r0 (boolean)
_feq	A==B	r0 (float)	r1 (float)	r0 (boolean)
_dflt	(double)A	r0 (int)		r0/r1 (double)
_dfltu	(double)A	r0 (unsigned)		r0/r1 (double)
_dfix	(int)A	r0/r1 (double)		r0 (int)
_dfixu	(unsigned)A	r0/r1 (double)		r0 (unsigned)
_fflt	(float)A	r0 (int)		r0 (float)
_ffltu	(float)A	r0 (unsigned)		r0 (float)
_ffix	(int)A	r0 (float)		r0 (int)
_ffixu	(int)A	r0 (float)		r0 (unsigned)
_f2d	(double)A	r0 (float)		r0/r1 (double)
_d2f	(float)A	r0/r1 (double)		r0 (float)

11.2.4 Type formats

`int, unsigned` 32-bit integer quantities.

`boolean` either 0 (False) or 1 (True)

`float` IEEE single precision floating-point number. (See Figure 11-1)

`double` IEEE double precision floating-point number. (See Figure 11-2)

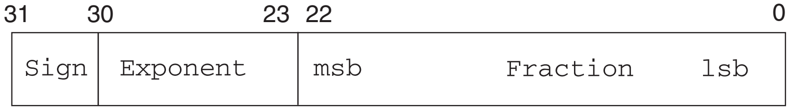


Figure 11-1 IEEE single precision floating-point format

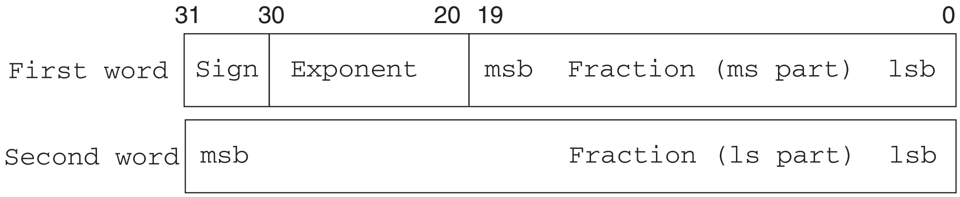


Figure 11-2 IEEE double precision floating-point format

11.3 Floating-point instructions

The ARM assembler supports a comprehensive floating-point instruction set. Floating-point operations are performed to the IEEE 754 standard. There are eight floating-point registers, numbered f0 to f7. Floating-point operations, like integer operations, are performed between registers.

Note

Floating-point operations are only usable from armcc and in ARM assembly language modules. They cannot be used from Thumb code because Thumb does not support the coprocessor instructions.

Precision must be specified for many floating-point operations where shown as *prec* below. The options are:

`S`	single
`D`	double
`E`	extended
`P`	packed BCD (only available for `LDF` and `STF` instructions).

In the following instruction patterns, *`round`* represents the rounding mode. It defaults to *round to nearest*. It can be set in the appropriate instructions to:

`P`	round to +infinity
`M`	round to –infinity
`Z`	round to zero.

In the following instruction patterns, `R`*`x`* represents an ARM register, and `F`*`x`* a floating-point register.

11.3.1 Floating-point data transfer: LDF and STF

`LDF`	load data to floating-point register
`STF`	store data from floating-point register

The syntax of these instructions is:

```
opcode{cond}prec Fd,[Rn,#offset]{!}
                    [Rn]{,#offset}
                    program-or-register-relative-expression
```

The memory address can be expressed in one of three ways, as shown above. In the first, pre-indexed form, an ARM register `Rn` holds the base address, to which an offset can be added if necessary. Writeback of the effective address to `Rn` can be enabled using `!` The offset must be divisible by `4`, and within the range –1020 to 1020 bytes.

With the second, post-indexed form, writeback of `Rn`+*`offset`* to `Rn` after the transfer is automatic, and the data is transformed from address `Rn`, not address `Rn` plus offset. Alternatively, a program-relative or register-relative expression can be used, in which case the assembler generates a pc-relative or register-relative, pre-indexed address. If it is out of range an error results.

11.3.2 Floating-point register transfer: FLT and FIX

```
FLT    integer to floating-point transfer      Fn := Rd
```

The syntax of this instruction is:

```
FLT{condition}prec{round} Fn,Rd
```

where `Rd` is an ARM register.

```
FIX    floating-point to integer transfer      Rd := Fn
```

The syntax of this instruction is:

```
FIX{condition}{round} Rd,Fn
```

11.3.3 Floating-point register transfer: status and control

The following instructions transfer values between the status and control registers of the floating-point coprocessor, and an ARM general purpose register. The syntax of the instructions is:

```
opcode{condition} Rd
```

The instructions are:

```
WFS    write floating-point status       FPSR:= Rd
RFS    read floating-point status        Rd:=FPSR
WFC    write floating-point control register
                           FPCR:= Rd (privileged modes only)
RFC    read floating-point control register
                           Rd:=FPCR (privileged modes only)
```

`WFC` and `RFC` should never be used by code outside the floating-point system (that is, the FPA and FPASC). They are only documented here for completeness.

11.3.4 Floating-point multiple data transfer: LFM and SFM

The load and store multiple floating point instructions are:

`LFM`	load floating-point multiple
`SFM`	store floating-point multiple

These instructions are used for block data transfers between the floating-point registers and memory. Values are transferred in an internal 96-bit format, with no loss of precision and with no possibility of an IEEE exception occurring, (unlike STFE which may fault on storing a trapping `NaN`).

There are two forms, depending on whether the instruction is being used for stacking operations or not. The first, nonstacking, form is:

```
opcode{condition} Fd,count,[Rn]
                           [Rn,#offset]{!}
                           [Rn],#offset
```

The first register to transfer is `F`*`d`*, and the number of registers to transfer is *`count`*. Up to four registers can be transferred, always in ascending order. The count wraps round at f7, so if f6 is specified with four registers to transfer, f6, f7, f0, and f1 will be transferred in that order.

With pre-indexed addressing, the destination/source register can be specified with or without an *`offset`* expressed in bytes. Writeback of the effective address to `R`*`n`* can be specified with `!`.

With post-indexed addressing (the third form above), writeback is automatically enabled, and the data is transferred from address R*n*, not (R*n* plus offset). Note that r15 cannot be used with writeback, and that *offset* must be divisible by 4 and in the range –1020 to 1020, as for other coprocessor loads and stores.

The second form adds a two-letter stacking mnemonic (below *ss*) to the instruction and optional condition codes. The mnemonic `FD` denotes a full descending stack (pre-decrement push, post-increment pop), while `EA` denotes an empty ascending stack (post-increment push, pre-decrement pop). The syntax is as follows:

```
opcode{condition}ss Fd,count,[Rn]{!}
```

`FD` and `EA` define pre-indexing and post-indexing, and the up/down bit by reference to the form of stack required. Unlike the integer block-data transfer operations, only `FD` and `EA` stacks are supported. The character `!`, if present, enables writeback of the updated base address to R*n*. r15 cannot be the base register if writeback is enabled.

The possible combinations of mnemonics are listed below:

`LFMFD` load floating-point multiple from a full descending stack (post-increment load)

`LFMEA` load floating-point multiple from an empty ascending stack (pre-decrement load)

`SFMFD` store floating-point multiple to a full descending stack (pre-decrement store)

`SFMEA` store floating-point multiple to an empty ascending stack (post-increment store)

11.3.5 Floating-point comparisons: CMF and CNF

The following instructions provide floating-point comparisons. The syntax of the instructions is:

```
opcode{condition} Fn,Fm
```

The instructions are:

CMF	compare floating-point	compare *Fn* with *Fm*CMFE
CNF	compare negated floating-point	compare *Fn* with -*Fm*CNFE

`CMF` and `CNF` only raise exceptions for signalling `NaN` operands and should be used to test for equality (Z clear/set) and unorderedness (V set/clear). To comply with IEEE 754-1985, all other tests should use `CMFE` or `CNFE`, which may raise an exception if either of the operands is any sort of `NaN`.

11.3.6 Floating-point binary operations

The following instructions provide floating-point binary operations. The syntax of the instructions is:

binop{*condition*}*prec*{*round*} *Fd,Fn,Fm*

where:

`Fm` Can be either a floating-point register, or one of the floating-point constants, #0, #1, #2, #3, #4, #5, #10, or #0.5.

Fast operations produce results that may only be accurate to single precision.

The instructions are:

ADF	add	Fd:=Fn+Fm
MUF	multiply	Fd:=Fn*Fm
SUF	subtract	Fd:=Fn–Fm
RSF	reverse subtract	Fd:=Fm–Fn
DVF	divide	Fd:=Fn/Fm
RDF	reverse divide	Fd:=Fm/Fn
POW	power	Fd:=Fn to the power of Fm
RPW	reverse power	Fd:=Fm to the power of Fn
RMF	remainder	Fd:=remainder of Fn/Fm
FML	fast multiply	Fd:=Fn*Fm
FDV	fast divide	Fd:=Fn/Fm
FRD	fast reverse divide	Fd:=Fm/Fn
POL	polar angle	Fd:=polar angle of Fn,Fm (=ATN(Fm/Fn) whenever the quotient exists)

11.3.7 Floating-point unary operations

The following instructions provide floating-point unary operation. The syntax of the instructions is:

unop{*condition*}*prec*{*round*} F*d*,F*m*

where:

F*m* can be either a floating-point register or one of the floating-point constants #0, #1, #2, #3, #4, #5, #10, or #0.5.

The instructions are:

MVF	move	F*d*:=F*m*
MNF	move negated	F*d*:=–F*m*
ABS	absolute value	F*d*:=ABS(F*m*)
RND	round to integral value	F*d*:=integer value of F*m* (using current rounding mode)
URD	unnormalized round	F*d*:= integer value of F*m*, possibly in abnormal form
NRM	normalize	F*d*:= normalised form of F*m*
SQT	square root	F*d*:=square root of F*m*
LOG	logarithm to base 10	F*d*:=log F*m*
LGN	logarithm to base e	F*d*:=ln *Fm*
EXP	exponent	F*d*:=eFm
SIN	sine	F*d*:=sine of F*m*
COS	cosine	F*d*:=cosine of F*m*
TAN	tangent	F*d*:=tangent of F*m*
ASN	arc sine	F*d*:=arc sine of F*m*
ACS	arc cosine	F*d*:=arc cosine of F*m*
ATN	arc tangent	F*d*:=arc tangent of F*m*

11.4 Configuring the FPA support code for a new environment

For information on how to configure the FPASC for a new environment, see Application Note 10: *Configuring the FPA Support Code/FPE* (ARM DAI 0040).

Note

This application note also discusses configuring floating-point emulation (FPE). The linkable FPE library is no longer supported by the ARM Software Development Toolkit.

11.5 Controlling floating-point exceptions

Both the /hardfp and /softfp modes provide a function, called __fp_status(), for setting and reading the status of either the FPA or the floating-point library.

Example 11-1 is an extract from stdlib.h:

Example 11-1

```
extern unsigned int __fp_status(unsigned int /* mask */,unsigned int /*flags*/);

#define __fpsr_IXE0x100000      /* inexact exception trap enable bit */
#define __fpsr_UFE0x80000       /* underflow exception trap enable bit */
#define __fpsr_OFE0x40000       /* overflow exception trap enable bit */
#define __fpsr_DZE0x20000       /* divide by zero exception trap enable bit */
#define __fpsr_IOE0x10000       /* invalid operation exception trap enable bit */
#define __fpsr_IXC0x10          /* inexact exception flag bit */
#define __fpsr_UFC0x8           /* underflow exception flag bit */
#define __fpsr_OFC0x4           /* overflow exception flag bit */
#define __fpsr_DZC0x2           /* divide by zero exception flag bit */
#define __fpsr_IOC0x1           /* invalid operation exception flag bit */
```

mask and flags are bitfields that correspond directly to the floating-point status register (FPSR) in the FPA and the floating-point library.

The function __fp_status() returns the current value of the status register, and also sets the writable bits of the word (the exception control and flag bytes) to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each status register bit, determined by the respective bits in mask and flags. These are shown in Table 11-2.

Table 11-2 Status register bit operations

mask bit	flags bit	effect
0	0	no effect
0	1	toggle bit in status register
1	0	clear bit in status register
1	1	set bit in status register

11.5.1 Return value

The `__fp_status()` function returns a one word (four byte) result that contains the current value of the status register, before any changes are applied.

Initially all exceptions are enabled, and no flags are set.

System ID byte

Bits 31:24 contain a system ID byte. The currently defined values are:

0x00	pre-FPA floating-point emulator
0x01	FPA compatible floating-point emulator
0x40	floating-point library
0x80	FPPC (obsolete)
0x81	FPA10 (with FPASC module)

The top bit (bit 31) is used to distinguish between hardware and software systems, and bit 30 is used to distinguish between software emulators and software libraries.

Exception trap enable byte

Each bit of the exception trap enable byte corresponds to one type of floating-point exception.

Note

The current floating-point library never produces those exceptions marked with a *. A bit in the cumulative exception flags byte is set as a result of executing a floating-point instruction only if the corresponding bit is not set in the exception trap enable byte. If the corresponding bit in the exception trap enable byte is set, a runtime error occurs (`SIGFPE` is raised in a C environment).

Bits 23:16 control the enabling of exceptions on floating-point errors:

bits 23:21 reserved

bit 20 (IXE) inexact exception enable*

bit 19 (UFE) underflow exception enable*

bit 18 (OFE) overflow exception enable

bit 17 (DZE) divide by zero exception enable

bit 16 (IOE) invalid operation exception enable

A set bit causes the system to take an exception trap if an error occurs. Otherwise a bit is set in the cumulative exception flags (see Figure 11-3) and the IEEE defined result is returned.

7...5	4	3	2	1	0
Reserved	IXC	UFC	OFC	DZC	IOC

Figure 11-3 Cumulative exception flags byte

System control byte

This byte is not used in the floating-point library system. Refer to the FPA datasheet for details of its meaning under FPA.

In particular, the NaN exception control bit (bit 9) is not supported by the floating-point library.

Exception flags byte

Bits 7:0 contain flags for whether each exception has occurred in the same order as the exception trap enable byte (see Figure 11-4). Exceptions occur as defined by IEEE 754.

23 22 21	20	19	18	17	16
Reserved	IXE	UFE	OFE	DZE	IOE

Figure 11-4 Exception trap enable byte

11.5.2 Example

Example 11-2 gives four examples of using `__fp_status()`.

Example 11-2

```
status = __fp_status(0,0);
/* reads the status register, does not change it */

__fp_status(__fpsr_DZE,0);
/* disable divide-by-zero exception trap */

overflow = __fp_status(__fpsr_OFC,0) & __fpsr_OFC;
/* read (and clear) overflow exception flag bit */

/* Report the type of floating-point system being used. */
switch (flags=(__fp_status(0,0)>>24))
    {
    case 0x0: case 0x1:
        printf("Software emulation\n");
        break;
    case 0x40:
        printf("Software library\n");
        break;
    case 0x80: case 0x81:
        printf("Hardware\n");
        break;
    default:
        printf("Unknown ");
        if (flags & (1<<7))
            printf("hardware\n");
        else
            printf("software %s\n",
                 flags & (1<<6) ? "library" : "emulation");
        break;
    }
```

Chapter 12
ARMulator

This chapter gives reference information about the ARMulator. It contains the following sections:

12.1 About the ARMulator

The ARMulator is a program that emulates the instruction sets and architecture of various ARM processor cores. It provides an environment for the development of ARM-targeted software on the supported workstation and PC host systems.

The ARMulator is instruction-accurate. That is, it models the instruction set and counts cycles accurately. As a result, it is well suited to software development and benchmarking of ARM-targeted software. It is not a cycle-accurate model because it does not model the precise timing characteristics of processors.

The ARMulator also supports a full ANSI C library to enable complete C programs to run on the emulated system. Refer to Chapter 4 *The C and C++ Libraries* for more information on C library support. See also Chapter 13 *Angel* in the *ARM Software Development Toolkit User Guide* for information on the C library semihosting SWIs supported by ARMulator.

The ARMulator is transparently connected to the ARM debuggers to provide a hardware-independent ARM software development environment. Communication takes place across the Remote Debug Interface (RDI). You can supply models written in C that interface to the external interface of the ARMulator.

This chapter provides details of the functions you can use to write your own models. For additional information about the ARMulator, refer to the *ARM Software Development Toolkit User Guide*.

12.2 Modeling an ARM-based system

A minimal ARMulator environment consists of:

Remote Debug Interface (RDI)
This is the interface between the ARMulator and its host debugger.

ARM Core model This is the model of an ARM processor such as the ARM6, ARM710, or StrongARM.

Memory interface This is the interface between the ARM core and the memory model. Refer to *The memory interface* on page 12-13 for more information.

Memory model This is the model of the memory system for a specific ARM core model. Map files can support simple memory only, other memory models can support memory mapped I/O. Refer to *Memory model interface* on page 12-16 for more information on memory model interface functions.

Additional ARMulator models are used to model coprocessors, interface to host operating systems, and extend ARMulator functionality. These models include:

Basic These can be used to add functionality to the ARMulator. There are two types of basic model, *early* and *late*. Refer to *Basic model interface* on page 12-7 for detailed information.

Veneer memory Additional memory models, called *veneer* memory models, can be installed between the ARM Core model and the default memory model. Refer to *Installing a veneer memory model* on page 12-8 for more information.

Coprocessor These model ARM coprocessors. Refer to *Coprocessor model interface* on page 12-24 for information on coprocessor model interface functions.

Operating System These provide a virtual interface between the host and the ARM model. Refer to *Operating system or debug monitor interface* on page 12-36 for more information on O/S model interface functions.

The Remote Debug Interface and ARM Core models are built into the ARMulator. You can add your own basic, memory, veneer memory, coprocessor, and operating system models.

12.2.1 Model stubs

Basic models, memory models, coprocessor models, and operating system models attach to the ARMulator through a stub. This stub consists of an initialization function and a textual name for the model, which the ARMulator uses to locate it. Some samples are provided in the rebuild kit to help you implement new models.

You can attach any number of models to an ARMulator without modifying existing models. You select the model to use when you run the ARMulator, and you do not need to recompile.

At startup, the ARMulator locates the model, then calls the initialization function, passing in a pointer to a structure containing a list of pointers that the model should fill in with implementation functions. The model should also register an `ExitUpcall()` (see *ExitUpcall* on page 12-59) during initialization, to free any state it sets up.

Model initialization sequence

The model initialization functions are called in the following order:

1. ARMulator Core model.
2. Early basic models (see *Basic model interface* on page 12-7).
3. Memory models, including veneer memory models installed by an early basic model (see *Basic model interface* on page 12-7)
4. Coprocessor models.
5. Operating system models.
6. Late basic models (see *Basic model interface* on page 12-7).

12.2.2 The ARMul_State state pointer

`ARMul_State` is an opaque data type that is a handle to the internal state of the ARMulator. All the models are passed a *`state`* variable of type `ARMul_State`. The ARMulator exports a number of functions to enable you to access ARMulator state **`struct`** members. See *Accessing ARMulator state* on page 12-42 for more information.

12.2.3 Handling armsd map files

The ARMulator does not directly support `armsd.map` files. However, a memory model can intercept the `RDIMemory_Access`, `RDIMemory_Map`, and `RDIInfo_Memory_Stats` RDI messages, and implement this functionality directly.

The sample model `armmap.c` does this, and implements a basic memory system that inserts wait states according to the memory speeds specified in the `armsd.map` file. Refer to the ARMulator chapter in the *ARM Software Development Toolkit User Guide* for more information on map files.

12.2.4 Configuring models through ToolConf

ARMulator models are configured through the `ToolConf`. The `ToolConf` is a database of tags and values that the ARMulator reads from a configuration file (`armul.cnf`) during initialization. The configuration file is documented in Application Note 52, *The ARMulator Configuration File* (ARM DAI 0052).

A number of functions are provided for looking up values from this database. The full set of functions is defined in `toolconf.h`. All the functions take an opaque handle called a `toolconf`.

The most frequently used functions are described below.

12.2.5 ToolConf_Lookup

This function performs a lookup on a specified tag in the `armul.cnf` database. If the tag is found, its associated value is returned. Otherwise, NULL is returned.

Syntax

```
const char *ToolConf_Lookup(toolconf hashv, const char *tag)
```

where:

`hashv` is the `armul.cnf` database to perform the lookup on.

`tag` is the tag to search for in the database. The tag is case-dependent.

Return

The function returns:

- a **`const`** pointer to the tag value, if the search is successful
- NULL, if the search is not successful.

Example

```
const char *option = ToolConf_Lookup(db, ARMulCnf_Size);

/* ARMulCnf_Size is defined in armcnf.h */
```

12.2.6 ToolConf_Cmp

This function performs a case-insensitive comparison of two ToolConf database tag values.

Syntax

```
int ToolConf_Cmp(const char *s1, const char *s2)
```

where:

s1 is a pointer to the first string value to compare.

s2 is a pointer to the second string value to compare.

Return

The function returns:

- 1, if the strings are identical
- 0, if the strings are different.

Example

```
if (ToolConf_Cmp(option, "8192"))
```

12.3 Basic model interface

The simplest model interface is the Basic model. This provides a mechanism for calling a user-supplied function during initialization (see *Basic model initialization function* on page 12-10). The function can then install upcalls, for example, to add functionality.

Basic models can be initialized either before or after memory models are initialized. This means that there are two distinct types of basic model:

- Early models
- Late models.

Whether a basic model is early or late is controlled by the location of its configuration in the ARMulator configuration file. See Application Note 52, *The ARMulator Configuration File* (ARM DAI 0052) for more information.

12.3.1 Late basic models

Late basic models are initialized after the memory models. They can call the memory system, and can, for example, initialize the memory contents. The `pagetable.c` model is an example of a late basic model. It writes an MMU pagetable to memory, after the memory system and MMU have been initialized.

12.3.2 Early basic models

Early basic models are initialized before memory models and can change the way the memory interface is initialized, primarily through calling `ARMul_InstallMemoryInterface()` (see *ARMul_InstallMemoryInterface* on page 12-11). In particular, early basic models can be used to install additional, *veneer* memory models.

Early models must not call the memory system (for example, `ARMul_WriteWord()`), because it is not initialized when the early model is called.

The `watchpnt.c` and `tracer.c` models are examples of early basic models. These models install watchpoint and trace veneer memory models. The following sections give more information on installing a veneer memory model.

Installing a veneer memory model

By default, the ARMulator initialization sequence installs the default memory model for a specific processor core. For example, Figure 12-1 shows the model hierarchy for an ARM7 after the memory model initialization function has completed.

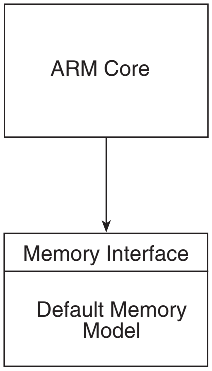


Figure 12-1 Minimal ARMulator model

You can use an early basic model to install any number of veneer memory models. The sequence of events is:

1. Define the early model in `armul.cnf`. The ARMulator calls the initialization function for the early model (see *Basic model initialization function* on page 12-10).

2. The early model initialization function must call `ARMul_InstallMemoryInterface()` to install the memory interface for the veneer memory model. This is required only if you are installing veneer memory models (see *ARMul_InstallMemoryInterface* on page 12-11).

3. When the initialization function for the early model returns, the ARMulator calls the memory model initialization function for the veneer memory model (see *Memory model initialization function* on page 12-17).

   The initialization function must call the initialization function for the model underneath it, either another veneer model or the standard memory model if there are no more veneer memory models installed.

4. When all veneer models are installed, the initialization function for the standard memory model for the processor model is called (*Memory model initialization function* on page 12-17).

Figure 12-2 shows an example of a model hierarchy with the watchpoint veneer installed.

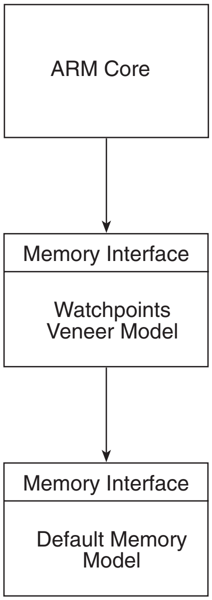


Figure 12-2 Veneer model hierarchy

12.3.3 Basic model initialization function

A basic model exports a function that is called during initialization. You must provide the model initialization function. If the model and the function are registered, and an `armul.cnf` entry is found, then the model initialization function is called.

The name of the function is defined by you. In the description below, the name `ModelInit` is used.

Syntax

```
static ARMul_Error ModelInit(ARMul_State *state,
                             toolconf config)
```

where:

state is the ARMulator state pointer.

config is the configuration database.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error during initialization.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

Example

The following example is from `watchpnt.c`.

```
#define ModelName (tag_t)"WatchPoints"

static ARMul_Error ModelInit(ARMul_State *state,
                             toolconf config)
{
    return ARMul_InstallMemoryInterface(state, TRUE, ModelName);
}

ARMul_ModelStub ARMul_WatchPointsInit = {
    ModelInit,
    ModelName
};
```

12.3.4 ARMul_InstallMemoryInterface

This function must be called from an early basic model that is installing a veneer memory model. It installs the memory interface for the veneer memory model.

Syntax

```
ARMul_Error ARMul_InstallMemoryInterface(ARMul_State *state,
                                         unsigned at_core,
                                         tag_t new_model)
```

where:

`state` is a pointer to the ARMulator state.

`at_core` indicates where to place the model:

`0`	places the model immediately above the lowest memory model in the memory hierarchy.
`non-zero`	places the model immediately below the processor.

`new_model` names the veneer memory model.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error during installation.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

Usage

This function must be called before the memory models are initialized, for example, from an early model (see *Early basic models* on page 12-7).

For a simple processor and memory system, *`at_core`* has no effect, because the lowest memory model is the one immediately below the processor. However, for a cached processor, a cache model sits between the processor and the lowest memory model, as shown in Figure 12-3 on page 12-12.

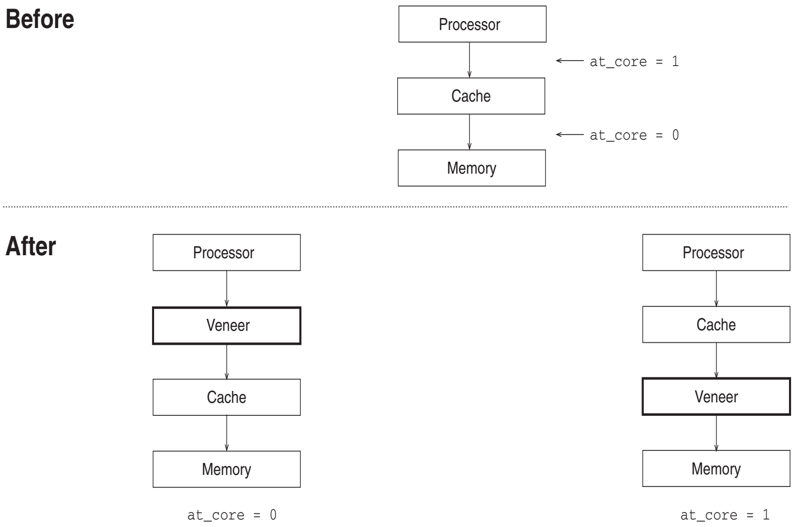


Figure 12-3 Inserting into a cache hierarchy

12.4 The memory interface

The memory interface is the interface between the ARMulator core and the memory model.

Because there are many core processor types, there are many memory type variants. The memory initialization function is told which type it should provide (see *Memory model initialization function* on page 12-17). A model must refuse to initialize in the case of an unrecognized memory type variant.

If you install a veneer memory model between the default memory model and the ARM core, you must explicitly install the memory interface for the veneer model by calling `ARMul_InstallMemoryInterface()`. See *ARMul_InstallMemoryInterface* on page 12-11 and *Installing a veneer memory model* on page 12-8 for more information.

Note

The **nTRANS** signal from the processor is not passed to the memory interface. Because this signal changes infrequently and might not be used by a memory model, a model should use `TransChangeUpcall()` to track **nTRANS** (see *TransChangeUpcall* on page 12-61).

12.4.1 Memory type variants

The memory type variants are defined in the `ARMul_MemInterface` structure in `armmem.h`. They are described in the following sections.

Basic memory types

There are three basic variants of memory type. All three use the same function interface to the core. The types are defined as follows:

`ARMul_MemType_Basic`

supports byte and word loads and stores.

`ARMul_MemType_16Bit`

is the same as `ARMul_MemType_Basic` but with the addition of halfword loads and stores.

`ARMul_MemType_Thumb`

is the same as `ARMul_MemType_16Bit` but with halfword instruction fetches (that can be sequential). This can indicate to a memory model that *most* accesses will be halfword-instruction-sequential rather than the usual word-instruction-sequential.

Note

Memory models that do not support halfword accesses should refuse to initialize for `ARMul_MemType_16Bit` and `ARMul_MemType_Thumb`.

For all three types, the model should fill in the `interf->x.basic` function pointers.

The file `armflat.c` contains an example function that implements a basic model. In `bytelane.c`, there is an example of a model of an ASIC that converts the basic memory types into the byte-lane version.

Cached versions of basic memory types

There are three variants of the basic memory types for cached processors such as the ARM710 and ARM740T. These variants are defined as follows:

- `ARMul_MemType_BasicCached`
- `ARMul_MemType_16BitCached`
- `ARMul_MemType_ThumbCached`

:These differ from the basic equivalents in that there are only two types of cycle:

- Memory cycle, where `acc_MREQ(acc)` is `TRUE`
- Idle cycle, where `acc_MREQ(acc)` is `FALSE`.

A non-sequential access consists of an Idle cycle followed by a Memory cycle, with the same `address` supplied for both.

A sequential access is a Memory cycle, with `address` incremented from the previous access.

Byte-lane memory for StrongARM

StrongARM variants are defined as follows:

- `ARMul_MemType_StrongARM`
- `ARMul_MemType_ByteLanes`

Externally, StrongARM can use a byte-lane memory interface. There is a StrongARM variant of the basic memory type that handles this. All the function types are the same, and the model must still fill in the basic part of the `ARMul_MemInterface` structure, but the meaning of the `ARMul_acc` word passed to the `access()` function is different.

The StrongARM variant replaces `acc_WIDTH` (see *armul_MemAccess* on page 12-21) with `acc_BYTELANE(acc)`. This returns a four-bit mask of the bytes in the word passed to the `access()` function that are valid.

There is no endianness problem with this method of access. The model can ignore endianness. Bit 0 of this word corresponds to bits 0-7 of the data, bit 1 to bits 8-15, bit 2 to bits 16-23, and bit 4 to bits 24-31.

ARM8 memory type

The ARM8 memory type is defined as follows:

- `ARMul_MemType_ARM8`

This is a double bandwidth interface. The ARM8 core can request two sequential accesses per cycle.

ARM9 memory type

The ARM9 memory type is defined as follows:

- `ARMul_MemType_ARM9`

The ARM9, 920, and 940 memory type has the same memory type for cached and non-cached cores. This is a Harvard architecture memory system.

Note

The data and instruction address space is common.

12.5 Memory model interface

The memory model interface is defined in the file `armmem.h` (which is `#included` from `armdefs.h`). All memory accesses are performed through a single function pointer that is passed a flags word. The flags word consists of a bitfield in which the bits correspond to the signals on the outside of the ARM processor. This determines the type of memory access that is being performed.

At initialization time, the initialization function registers a number of functions in the memory interface structure, `ARMul_MemInterface` in `armmem.h`. The basic entries are:

```
typedef struct armul_meminterface ARMulMemInterface;
struct armul_meminterface {
    void *handle;
    armul_ReadClock *read_clock;
    armul_ReadCycles *read_cycles;
    union {
      struct {
        armul_MemAccess *access;
        armul_GetCycleLength *get_cycle_length;
      } basic;
      // ... other processor specific entries follow
```

The following sections describe the initialization function and the basic function entries:

- *Memory model initialization function* on page 12-17
- *armul_ReadClock* on page 12-19
- *armul_GetCycleLength* on page 12-19
- *armul_ReadCycles* on page 12-20
- *armul_MemAccess* on page 12-21.

There are two functions that allow you to set and return the address of the top of memory. These are described in:

- *ARMul_SetMemSize* on page 12-23
- *ARMul_GetMemSize* on page 12-23.

12.5.1 Memory model initialization function

The memory model exports a function that is called during initialization. You must provide the memory model initialization function. If the model and the function are registered, and an `armul.cnf` entry is found, then the memory model initialization function is called.

The name of the function is defined by you. In the description below, the name `MemInit` is used.

Syntax

```
static ARMul_Error MemInit(ARMul_State *state,
                           ARMul_MemInterface *interf,
                           ARMul_MemType variant,
                           toolconf config)
```

where:

`state` is a pointer to the ARMulator state.

`interf` is a pointer to the memory interface structure. See the `ARMul_MemInterface` structure in `armmem.h` for an example.

`variant` is the memory interface variant. See the `ARMul_MemType` enumeration in `armmem.h`. Refer to *Memory type variants* on page 12-13 for a description of the variants.

`config` is the configuration database.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error during initialization.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

Usage

The initialization should set the handle for the model by assigning to `interf->handle`. The handle is usually a pointer to the state representing this instantiation of the model. It is passed to all the access functions called by the ARMulator. See also *ARMul_AddCounterDesc* and *ARMul_AddCounterValue* on page 12-80.

This function should also be used to:

- register any upcalls
- announce itself to the user using `ARMul_PrettyPrint()`.
- attach any associated coprocessor models (CP15, for example) and set up its state.

Example

Refer to the definition of `MemInit` in `armflat.c` for an example. `MemInit` installs `ReadClock()`, `ReadCycles()`, `MemAccess()`, and `GetCycleLength()` functions. Refer to the following sections for more information on implementing these functions:

- *armul_ReadClock* on page 12-19
- *armul_GetCycleLength* on page 12-19
- *armul_ReadCycles* on page 12-20
- *armul_MemAccess* on page 12-21.

12.5.2 armul_ReadClock

This function should return the elapsed time in µ-seconds since the emulation model reset.

The `read_clock` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_ReadClock()` function.

Syntax

```
unsigned long armul_ReadClock(void *handle)
```

where:

handle is a pointer to the ARMulator state.

Return

The function returns an unsigned long value representing the elapsed time in µ-seconds since the model reset.

Usage

A model can supply NULL if it does not support this functionality.

12.5.3 armul_GetCycleLength

The `get_cycle_length` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_GetCycleLength()` function. This function should return the length of a single cycle in units of one tenth of a nanosecond.

You should implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
unsigned long armul_GetCycleLength(void *handle)
```

where:

handle is a pointer to the ARMulator state.

Return

The function returns an unsigned long representing the length of a single cycle in units of one tenth of a nanosecond. For example, it returns 300 for a 33.3MHz clock.

12.5.4 armul_ReadCycles

The `read_cycles` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_ReadCycles()` function. This function should calculate the total cycle count since the emulation model reset. You should implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
const ARMul_Cycles *armul_ReadCycles(void *handle)
```

where:

`handle` is a pointer to the ARMulator state.

Return

The function is called each time the counters are read by the debugger. The function calculates the total cycle count and returns a pointer to the `ARMul_Cycles` structure that contains the cycle counts. The `ARMul_Cycles` structure is defined as:

```
typedef struct {unsigned long Total;
               unsigned long NumNcycles, NumScycles,
                             NumCcycles, NumIcycles, NumFcycles;
               unsigned long CoreCycles;
               } ARMul_Cycles;
```

Usage

A model can keep count of the accesses made to it by the ARMulator by providing this function. The value of the `CoreCycles` field in `ARMul_Cycles`, is provided by the ARMulator, not the memory model. When you write this function you must calculate the `Total` field, because this is the value returned when `ARMul_Time()` is called. See *Event scheduling functions* on page 12-70 for a description of `ARMul_Time()`.

These counters are also used to provide the `$statistics` variable inside the ARM debuggers, if the memory model does not use `ARMul_AddCycleDesc()` and `ARMul_AddCounterDesc()`. See *Upcalls* and *ARMulator specific functions* on page 12-75.

Example

```
ARMul_Cycles *cycles;
cycles = interf->read_cycles(handle);
// where interf is a pointer to the memory interface structure.
// and handle is a void * pointer to the ARMul_State structure.
```

12.5.5 armul_MemAccess

The `access` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_MemAccess()` function. This function is called on each ARM core cycle.

You must implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
int armul_MemAccess(void *handle, ARMword address,
                    ARMword *data, ARMul_acc access_type)
```

where:

`handle` is the value assigned to `interf->handle` in the initialization function.

`address` is the value on the address bus.

`data` is a pointer to the data for the memory access. Refer to the Usage section below for details.

`access_type`

encodes the type of cycle. On some processors (for example, cached processors) some of the signals will not be valid. The macros for determining access type are:

`acc_MREQ(acc)`

chooses between memory request and non-memory request accesses.

`acc_WRITE(acc) or acc_READ(acc)`

for memory cycles, determines whether this is a read or write cycle (NOT `acc_READ` implies `acc_WRITE`, and NOT `acc_WRITE` implies `acc_READ`).

`acc_SEQ(acc)`

for a memory cycle, this is `TRUE` if the address is the same as, or sequentially follows from the address of the preceding cycle. For a non-memory cycle it distinguishes between coprocessor (`acc_SEQ`) and idle (not `acc_SEQ`) cycles.

`acc_OPC(acc)`

for memory cycles, this is `TRUE` if the data being read is an instruction. (It is never `TRUE` for writes.)

`acc_LOCK(acc)`
: distinguishes a read-lock-write memory cycle.

`acc_ACCOUNT(acc)`
: is `TRUE` if the cycle is coming from the ARM core, rather than the remote debug interface.

`acc_WIDTH(acc)`
: returns `BITS_8`, `BITS_16`, or `BITS_32` depending on whether a byte, halfword, or word is being fetched/written on a data access.

Return

The function returns:

- 1, to indicate successful completion of the cycle.
- 0, to indicate that the processor should busy-wait and try the access again next cycle.
- –1, to signal an abort.

Usage

Reads For reads, the memory model function should write the value to be read by the core to the word pointed to by *`data`*. For example, with a byte load it should write the byte value, with a halfword load it should write the halfword value.

The model can ignore the alignment of the address passed to it because this is handled by the ARMulator. However, it must present the bytes of the word in the correct order for the endianness of the processor. This can be determined by using either a `ConfigChangeUpcall()` upcall or `ARMul_SetConfig()` (see *Accessing ARMulator state* on page 12-42).

`armdefs.h` provides a flag variable/macro named `HostEndian`, which is `TRUE` if the ARMulator is running on a big-endian machine. See the `armflat.c` sample file for an example of how to handle endianness.

Writes For writes, *`data`* points to the datum to be stored. However, this value may need to be shortened for a byte or halfword store.

As with reads, endianness must be handled correctly.

12.5.6 ARMul_SetMemSize

This function should be called during memory initialization. It specifies the size, and therefore the top of memory.

Syntax

```
ARMword ARMul_SetMemSize(ARMul_State *state, ARMword size)
```

where:

state is a pointer to the ARMulator state.

size is the size of memory in bytes (word aligned).

Return

The function returns the previous `MemSize` value.

Usage

The value of `size` should not exceed 0x80000000.

12.5.7 ARMul_GetMemSize

This function returns the address of the top of memory.

Syntax

```
ARMword ARMul_GetMemSize(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the highest available address in memory.

Usage

This function can be used, for example, by a debug monitor model to tell an application where the top of usable memory is, so it can set up application memory.

12.6 Coprocessor model interface

The coprocessor model interface is defined in `armdefs.h`. The basic coprocessor functions are:

- *init* on page 12-28
- *ldc* on page 12-29
- *stc* on page 12-30
- *mrc* on page 12-31
- *mcr* on page 12-32
- *cdp* on page 12-33.

In addition, two functions are provided that enable a debugger to read and write coprocessor registers through the Remote Debug Interface. They are:

- *read* on page 12-34
- *write* on page 12-35.

If a coprocessor does not handle one or more of these functions, it should leave their entries in the `ARMul_CPInterface` structure unchanged.

12.6.1 The ARMul_CPInterface structure

The coprocessor initialization structure contains a set of function pointers for each supported operation. You must use the coprocessor initialization function to install your functions in the structure at initialization. Refer to *init* on page 12-28 for more information.

This structure also contains a pointer to a `reg_bytes` array that contains:

- the number of coprocessor registers, in the first element
- the number of bytes available to each register, in the remaining elements.

For example, `dummymmu.c` defines an array of eight registers, each of four bytes:

```
static const unsigned int MMURegBytes[] = {8, 4,4,4,4,4,4,4,4};
```

Definition

The `ARMul_CPInterface` structure is defined as:

```
typedef struct ARMul_cop_interface_str ARMul_CPInterface;

struct ARMul_cop_interface_str {
    void *handle;                   /* A model private handle */
    armul_LDC *ldc;                 /* LDC instruction */
    armul_STC *stc;                 /* STC instruction */
    armul_MRC *mrc;                 /* MRC instruction */
    armul_MCR *mcr;                 /* MCR instruction */
    armul_CDP *cdp;                 /* CDP instruction */
    armul_CPRead *read;             /* Read CP register */
    armul_CPWrite *write;           /* Write CP register */
    const unsigned int *reg_bytes;  /* map of CP reg sizes */
}
```

Example

In Example 12-1 on page 12-26, if the core has a memory management unit (MMU), a predefined `mmu->RegBytes` is used. If the core has a protection unit (PU), the size of `RegBytes[7]` and `RegBytes[8]` is modified. `RegBytes[7]` corresponds to CP register 6 and `RegBytes[8]` corresponds to CP register 7.

CP register 6 is the protection region base/size register and has eight indexable registers, so it is set to size `sizeof(ARMword)*8`.

Refer to the *ARM Architectural Reference Manual* for more information on the MMU and PU.

Example 12-1

```
{
    if (mem->prop & Cache_ProtectionUnit_Prop)
        {
        /* Use the PU */
        mmu->RegBytes[0]=8;                  // has 8 registers
        mmu->RegBytes[7]=sizeof(ARMword)*8; // register 7 is 8 words long
        mmu->RegBytes[8]=sizeof(ARMword);   // register 8 is a single word
        interf->mrc=PU_MRC;
        interf->mcr=PU_MCR;
        interf->read=PU_CPRead;
        interf->write=PU_CPWrite;
        interf->reg_bytes=mmu->RegBytes;
        ARMul_PrettyPrint(state,", PU");



        /* Initialise PU Area registers to 0 */
        for ( i=0; i<=7; i++)
            {
            mmu->PU_Areas[i].PU_Register=0;
            }
        }
    else /* Use the MMU */
        {
        interf->mrc=MRC;
        interf->mcr=MCR;
        interf->read=CPRead;
        interf->write=CPWrite;
        interf->reg_bytes=mmu->RegBytes;
        ARMul_PrettyPrint(state,", MMU");
        }
}
```

12.6.2 ARMul_CoProAttach

Coprocessors are either initialized directly by the ARMulator as appropriate, or can be attached directly by another model by calling `ARMul_CoProAttach()`. As with memory models, the coprocessor initialization function is used to fill in the interface structure. `ARMul_CoProAttach()` registers the coprocessor initialization function for a specified processor. You must implement this function.

Syntax

```
ARMul_Error ARMul_CoProAttach(ARMul_State *state,
                              unsigned number,
                              const ARMul_CPInit *init,
                              toolconf config,
                              void *sibling)
```

where:

state is a pointer to the ARMulator state.

number is the coprocessor number to attach.

init is a pointer to a coprocessor initialization function.

config is the configuration database.

sibling is a pointer to the state to be shared with the coprocessor.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error during initialization.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

Example

```
error = ARMul_CoProAttach(state, 4, init, config, handle);
```

12.6.3 init

This is the coprocessor initialization function. This function fills in the `ARMul_CPInterface` structure for the coprocessor model (see *The ARMul_CPInterface structure* on page 12-24).

Syntax

```
ARMul_Error init(ARMul_State *state, unsigned num,
                ARMul_CPInterface *interf, toolconf config,
                void *sibling)
```

where:

`state` is a pointer to the ARMulator state.

`num` is the coprocessor number.

`interf` is a pointer to the `ARMul_CPInterface` structure to be filled in.

`config` is the configuration database.

`sibling` identifies associations between the coprocessor and other emulated components, such as sibling coprocessors. For example, a system may have a pair of coprocessors that must be aware of each other. This is the value passed to `ARMul_CoProAttach()`.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

12.6.4 ldc

This function is called when an `LDC` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned ldc(void *handle, unsigned type, ARMword instr,
             ARMword data)
```

where:

handle is a pointer to the ARMulator state.

type is the type of coprocessor access. This can be one of:

`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.

`ARMul_TRANSFER` requests transfer.

`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor should discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST.`

`ARMul_DATA` indicates that valid data is included in *data*.

instr the current opcode.

data is the data being transferred to the coprocessor.

Return

The function should return one of:

- `ARMul_INC`, to request more data from the core.
- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_BUSY`, to indicate that the coprocessor is busy.
- `ARMul_CANT`, to indicate that the instruction is not supported.

12.6.5 stc

This function is called when an `STC` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned stc(void *handle, unsigned type, ARMword instr,
             ARMword *data)
```

where:

`handle` is a pointer to the ARMulator state.

`type` is the type of the coprocessor access. This can be one of:

`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.

`ARMul_TRANSFER` requests transfer.

`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor should discard the current instruction. Usually, the instruction will be retried later, in which case the *`type`* will be reset to `ARMul_FIRST.`

`ARMul_DATA` indicates that valid data is included in *`data`*.

`instr` is the current opcode.

`data` is a pointer to the location of the data being transferred from the coprocessor to the core.

Return

The function should return one of:

- `ARMul_INC`, to indicate that there is more data to transfer to the core.
- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_BUSY`, to indicate that the coprocessor is busy.
- `ARMul_CANT`, to indicate that the instruction is not supported.

12.6.6 mrc

This function is called when an `MRC` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned mrc(void *handle, unsigned type, ARMword instr,
             ARMword *data)
```

where:

handle is a pointer to the ARMulator state.

type is the type of the coprocessor access. This can be one of:

`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.

`ARMul_TRANSFER` requests transfer.

`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor should discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST.`

`ARMul_DATA` indicates that valid data is included in *data*.

instr is the current opcode.

data is a pointer to the location of the data being transferred from the coprocessor to the core.

Return

The function should return one of:

- `ARMul_INC`, to indicate that there is more data to transfer.
- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_BUSY`, to indicate that the coprocessor is busy.
- `ARMul_CANT`, to indicate that the instruction is not supported.

12.6.7 mcr

This function is called when an `MCR` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned mcr(void *handle, unsigned type, ARMword instr,
             ARMword data)
```

where:

handle is a pointer to the ARMulator state.

type is the type of the coprocessor access. This can be one of:

`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.

`ARMul_TRANSFER` requests transfer.

`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor should discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST.`

`ARMul_DATA` indicates valid data is included in *data*.

instr is the current opcode.

data is the data being transferred to the coprocessor.

Return

The function should return one of:

- `ARMul_INC`, to indicate that there is more data to transfer.
- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_BUSY`, to indicate that the coprocessor is busy.
- `ARMul_CANT`, to indicate that the instruction is not supported.

12.6.8 cdp

This function is called when a CDP instruction is recognized for a coprocessor. If the requested coprocessor operation is not supported, the function should return ARMul_CANT. You should implement this function.

Syntax

unsigned cdp(**void** **handle*, **unsigned** *type*, ARMword *instr*)

where:

handle is a pointer to the ARMulator state.

type is the type of the coprocessor access. This can be one of:

ARMul_FIRST indicates that this is the first time the coprocessor model has been called for this instruction.

ARMul_INTERRUPT warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor should discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to ARMul_FIRST.

instr is the current opcode.

Return

The function should return one of:

- ARMul_DONE, to indicate that the coprocessor operation is complete.
- ARMul_BUSY, to indicate that the coprocessor is busy.
- ARMul_CANT, to indicate that the instruction is not supported.

12.6.9 read

This function enables a debugger to read a coprocessor register through the RDI. The function reads the coprocessor register numbered *`reg`* and transfers its value to the location addressed by *`value`*.

If the requested coprocessor register does not exist, or the register cannot be read, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned read(void *handle, unsigned reg, ARMword const *value)
```

where:

`handle` is a pointer to the ARMulator state.

`reg` is the register number of the coprocessor register to be read.

`value` is a pointer to the location of the data to be read from the coprocessor by RDI.

Return

The function should return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_CANT`, to indicate that the register is not supported.

Usage

This function can be useful for debugging purposes.

12.6.10 write

This function enables a debugger to write a coprocessor register through the RDI. The function writes the coprocessor register numbered *`reg`* with the value at the location addressed by *`value`*.

If the requested coprocessor does not exist or the register cannot be written, the function should return `ARMul_CANT`. You should implement this function.

Syntax

```
unsigned write(void *handle, unsigned reg, ARMword const *value)
```

where:

`handle` is a pointer to the ARMulator state.

`reg` is the register number of the coprocessor register that is to be written.

`value` is a pointer to the location of the data that is to be written to the coprocessor.

Return

The function should return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete.
- `ARMul_CANT`, to indicate that the register is not supported.

Usage

This function can be useful for debugging purposes.

12.7 Operating system or debug monitor interface

The ARMulator supports rapid prototyping of low level operating system code through an interface that enables a model to intercept SWIs and exceptions, and model them on the host. This model can communicate with the emulated application by reading and writing the emulated ARM state using the routines described in *Accessing ARMulator state* on page 12-42.

The interface functions are:

- *init* on page 12-37
- *handle_swi* on page 12-38
- *exception* on page 12-39.

These functions are described in more detail in the following sections.

12.7.1 The ARMul_OSInterface structure

The `ARMul_OSInterface` structure is defined as:

```
typedef struct armul_os_interface ARMul_OSInterface;

typedef ARMul_Error armul_OSInit(ARMul_State *state,
                                 ARMul_OSInterface *interf,
                                 toolconf config);
typedef unsigned armul_OSHandleSWI(void *handle,ARMword number);
typedef unsigned armul_OSException(void *handle, ARMword vector,
                                   ARMword pc);

struct armul_os_interface {
  void *handle;                  /* A model private handle */
  armul_OSHandleSWI *handle_swi;  /* SWI handler */
  armul_OSException *exception;   /* Exception handler */
};

typedef struct {
  armul_OSInit *init;            /* O/S initializer */
  tag_t name;                    /* O/S name */
} ARMul_OSStub;
```

12.7.2 init

This is the OS initialization function. It is passed a vector of functions to fill in. As with other models, the operating system model is called through an initialization function exported in a stub. You must implement this function.

The memory system is guaranteed to be operating at this time, so the operating system can read and write to the emulated memory using the routines described in *Memory access functions* on page 12-68.

Syntax

```
typedef ARMul_Error init(ARMul_State *state,
                         ARMul_OSInterface *interf,
                         toolconf config)
```

where:

`state` is a pointer to the ARMulator state.

`interf` is a pointer to the OS interface structure.

`config` is the configuration database.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error.
- An `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 12-75).

Usage

This function can also run initialization code.

12.7.3 handle_swi

This is the OS model SWI handling function. It is called whenever a `SWI` instruction is executed. This enables support code to simulate operating system operations. This code can model as much of your operating system as you choose. You should implement this function.

Syntax

`typedef unsigned` `handle_swi(`**`void`** `*`*`handle`*`, ARMword` *`number`*`)`

where:

`handle` is a pointer to the ARMulator state.

`number` is the SWI number.

Return

The function can refuse to handle the `SWI` by returning `FALSE`, or the model may choose not to handle `SWI` instructions by setting NULL as the `handle_swi` function. In either case, the SWI exception vector is taken by ARMulator. If the function returns `TRUE` the ARMulator continues from the next instruction after the `SWI`.

12.7.4 exception

This is the OS model exception handling function. It is called whenever an exception occurs. You should implement this function.

Syntax

```
typedef unsigned exception(void *handle, ARMword vector,
                           ARMword pc)
```

where:

handle is a pointer to the ARMulator state.

vector contains the address of the vector about to be executed, for example:

0x00	Reset
0x04	Undefined Instruction
0x1C	Fast Interrupt (FIQ).

pc contains the program counter (including the effect of pipelining) at the time the exception occurred.

Return

If the function returns TRUE, the ARMulator continues from the instruction following the instruction that was being executed when the exception occurred.

Note

If the exception is an interrupt, or a prefetch or data abort, the user function should make the ARMulator retry the instruction, rather than continuing from the following instruction. The user function can set up the pc by calling ARMul_SetPC to ensure this, before returning TRUE.

A return value of FALSE causes the ARMulator to handle the exception normally.

Usage

The CPU state is frozen immediately after the exception has occurred, but before the CPU has switched processor state or taken the appropriate exception vector.

12.8 Using the floating-point emulator (FPE)

The ARMulator is supplied with the floating-point emulator (FPE) in object form. The debug monitor model (`angel.c`) loads and starts executing the FPE on initialization.

The FPE requires the following SWIs to be supported by the debug monitor. Angel does not support these SWIs, however they are implemented by `angel.c` to support FPE:

- `SWI_Exit` (0x11)
- `SWI_GenerateError` (0x71)

To load and initialize the FPE, call the following functions:

- `ARMul_FPEInstall()`
- `ARMul_FPEVersion()`
- `ARMul_FPEAddressInEmulator()`.

These are described in more detail in the following sections.

12.8.1 ARMul_FPEInstall

This function writes the FPE into memory (below 0x8000), and executes it.

Syntax

`int` `ARMul_FPEInstall(ARMul_State *`*`state`*`)`

where:

`state` is a pointer to the ARMulator state.

Usage

Note

Because this involves running code, it must be done only after the ARMulator is fully initialized. Before calling `ARMul_FPEInstall()`, Angel completely initializes itself.

Return

The function returns:

- `TRUE`, if the installation is successful
- `FALSE`, if the installation fails.

12.8.2 ARMul_FPEVersion

This function returns the FPE version number. Angel uses this for unwinding aborts inside the emulator (see the `angel.c` source code for details).

Syntax

```
int ARMul_FPEVersion(ARMul_State *state)
```

where:

`state` is a pointer to the ARMulator state.

Return

The function returns either:

- the FPE version code, if available
- –1 if there is no FPE.

12.8.3 ARMul_FPEAddressInEmulator

This function returns `TRUE` if the specified address lies inside the emulator.

Syntax

```
int ARMul_FPEAddressInEmulator(ARMul_State *state, ARMword addr)
```

where:

`state` is a pointer to the ARMulator state.

`address` is the address to check.

Return

The function returns:

- `FALSE`, if there is no FPE, or the address is not in the FPE.
- `TRUE`, if the address is in the FPE.

12.9 Accessing ARMulator state

All the models are passed a `state` variable of type `ARMul_State`. This is an opaque handle to the internal state of the ARMulator. The ARMulator exports the following functions to enable models to access the ARMulator state through this handle:

Functions to access ARM registers

The following functions provide read and write access to ARM registers:

- *ARMul_GetMode* on page 12-44
- *ARMul_GetReg* on page 12-45
- *ARMul_SetReg* on page 12-46
- *ARMul_GetR15 and ARMul_GetPC* on page 12-47
- *ARMul_SetR15 and ARMul_SetPC* on page 12-48
- *ARMul_GetCPSR* on page 12-49
- *ARMul_SetCPSR* on page 12-50
- *ARMul_GetSPSR* on page 12-50
- *ARMul_SetSPSR* on page 12-51.

Functions to access coprocessor registers

The following functions call the read and write methods for a coprocessor:

- *ARMul_CPRead* on page 12-52
- *ARMul_CPWrite* on page 12-53.

Changing the model processor configuration

The following function enables you to change the configuration of your modeled processor:

- *ARMul_SetConfig* on page 12-54.

Note

It is not appropriate to access some parts of the state from certain parts of a model. For example, you should not set the contents of an ARM register from a memory access function, because the memory access function may be called during emulation of an instruction. In contrast, it is necessary to set the contents of ARM registers from a SWI handler function.

A number of the following functions take an **`unsigned`** mode parameter to specify the processor mode. The mode numbers are defined in `armdefs.h`, and are listed in Table 12-1.

In addition, the special value `CURRENTMODE` is defined. This enables `ARMul_GetMode()` to return the current mode number.

Table 12-1 Defined processor modes

`USER26MODE`	`USER32MODE`	`ABORT32MODE`
`FIQ26MODE`	`FIQ32MODE`	`UNDEF32MODE`
`IRQ26MODE`	`IRQ32MODE`	`SYSTEM32MODE`
`SVC26MODE`	`SCV32MODE`	

Note

26-bit mode is included here for backward compatibility only, and *will not be supported in future releases*.

For more information about 26-bit modes, see the *ARM Architectural Reference Manual*, and also refer to Application Note 11 *Differences Between ARM6 and Earlier ARM Processors*, and Application Note 37 *Startup configuration of ARM Processors with MMUs*.

12.9.1 ARMul_GetMode

This function returns the current processor mode.

Syntax

```
ARMword ARMul_GetMode(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Returns

This function returns the current mode.

See Table 12-1 on page 12-43 for a list of defined processor modes.

Usage

If this is to be done frequently, a model should install a `ModeChange()` upcall instead (see *ModeChangeUpcall* on page 12-60).

12.9.2 ARMul_GetReg

This function reads a register for a specified processor mode.

Syntax

```
ARMword ARMul_GetReg(ARMul_State *state, unsigned mode,
                     unsigned reg)
```

where:

state is a pointer to the ARMulator state.

mode is the current processor mode. Values for mode are defined in `armdefs.h` (see Table 12-1 on page 12-43)

reg is the register number of the register to read.

Return

The function returns the value in the given register for the specified mode.

Usage

Note

Register r15 should not be accessed with this function. Use `ARMul_GetPC()` or `ARMul_GetR15()` as described in *ARMul_GetR15 and ARMul_GetPC* on page 12-47.

12.9.3 ARMul_SetReg

This function writes a register for a specified processor mode.

Syntax

```
void ARMul_SetReg(ARMul_State *state, unsigned mode,
                  unsigned reg, ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode. Mode numbers are defined in `armdefs.h` (see Table 12-1 on page 12-43).

reg is the register number of the register to write.

value is the value to be written to register `reg` for the specified processor mode.

Usage

Note

Register r15 should not be accessed with this function. Use `ARMul_SetPC()`, or `ARMul_SetR15()` as in *ARMul_SetR15 and ARMul_SetPC* on page 12-48.

12.9.4 ARMul_GetR15 and ARMul_GetPC

The following functions read register r15.

Syntax

```
ARMword ARMul_GetR15(ARMul_State *state)
ARMword ARMul_GetPC(ARMul_State *state)
```

where:

`state` is a pointer to the ARMulator state.

Return

The functions return the value of register r15. If the processor is in a 32-bit mode the effect of either variant is the same.

If the processor is in a 26-bit mode:

- The `ARMul_GetPC` variant returns only the pc part of r15. It does not return the condition code and mode bits from register r15.
- The `ARMul_GetR15` variant returns the whole value of r15, including the condition code and mode bits.

Usage

Note

26-bit mode is included here for backward compatibility only, and *will not be supported in future releases*.

For more information about 26-bit modes, see the *ARM Architectural Reference Manual*, and also refer to Application Note 11 *Differences Between ARM6 and Earlier ARM Processors*, and Application Note 37 *Startup configuration of ARM Processors with MMUs*.

12.9.5 ARMul_SetR15 and ARMul_SetPC

The following functions write register r15.

Syntax

```
void ARMul_SetR15(ARMul_State *state, ARMword value)
void ARMul_SetPC(ARMul_State *state, ARMword value)
```

where:

state	is a pointer to the ARMulator state.
value	the new value of r15 (pc) to be written.

Return

The functions write a value into register r15. If the processor is in a 32-bit mode the effect of either variant is the same.

If the processor is in a 26-bit mode:

- The `ARMul_SetPC` variant sets only the pc part of r15. It leaves the condition code and mode bits unaltered.
- The `ARMul_SetR15` variant sets the whole of r15, including the condition code and mode bits.

Usage

Note

26-bit mode is included here for backward compatibility only, and *will not be supported in future releases*.

For more information about 26-bit modes, see the *ARM Architectural Reference Manual*, and also refer to Application Note 11 *Differences Between ARM6 and Earlier ARM Processors*, and Application Note 37 *Startup configuration of ARM Processors with MMUs*.

12.9.6 ARMul_GetCPSR

This function reads the CPSR. A valid value is faked if the processor is in a 26-bit mode, so this function can still be used.

Syntax

```
ARMword ARMul_GetCPSR(ARMul_State *state)
```

where:

`state` is a pointer to the ARMulator state.

Return

The function returns the value of the CPSR for the current mode.

Usage

Note

26-bit mode is included for backward compatibility only and *will not be supported in future releases*.

12.9.7 ARMul_SetCPSR

This function writes a value to the CPSR for the current processor mode. A valid value is faked if the processor is in a 26-bit mode, so this function can still be used.

Syntax

```
void ARMul_SetCPSR(ARMul_State *state, ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode. Values for mode are defined in `armdefs.h` (see Table 12-1 on page 12-43).

value is the value to be written to the CPSR for the current mode.

Usage

Note

26-bit mode is included for backward compatibility only and *will not be supported in future releases*.

12.9.8 ARMul_GetSPSR

This function reads the SPSR for a specified processor mode.

Syntax

```
ARMword ARMul_GetSPSR(ARMul_State *state, ARMword mode)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode for the SPSR to be read.

Return

The function returns the value of the SPSR for the specified mode.

12.9.9 ARMul_SetSPSR

This function writes the SPSR for a specified processor mode.

Syntax

```
void ARMul_SetSPSR(ARMul_State *state, ARMword mode,
                   ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode for the SPSR to be read. Values for mode are defined in `armdefs.h` (see Table 12-1 on page 12-43).

value is the new value to be written to the SPSR for the specified mode.

12.9.10 ARMul_CPRegBytes

This function returns the `reg_bytes[]` array for the specified coprocessor (see *The ARMul_CPInterface structure* on page 12-24 for details).

Syntax

```
unsigned int const *ARMul_CPRegBytes(ARMul_State *state,
                                     unsigned CPnum)
```

where:

state is a pointer to the ARMulator state.

CPnum is the coprocessor number to return the `reg_bytes[]` array for.

12.9.11 ARMul_CPRead

This function calls the `read` method for a coprocessor. It also intercepts calls to read the FPE emulated registers. (See *Using the floating-point emulator (FPE)* on page 12-40.)

Syntax

```
unsigned ARMul_CPRead(void *handle, unsigned reg,
                      ARMword *value)
```

where:

handle is a pointer to the ARMulator state.

reg is the number of the coprocessor register to read from.

value is the address to write the register value to.

Return

The function should return:

- `ARMul_DONE`, if the register can be read
- `ARMul_CANT`, if the register cannot be read.

12.9.12 ARMul_CPWrite

This function calls the `write` method for a coprocessor. It also intercepts calls to write the FPE emulated registers. (See *Using the floating-point emulator (FPE)* on page 12-40.)

Syntax

```
unsigned ARMul_CPWrite(void *handle, unsigned reg
                       ARMword const *value)
```

where:

handle is a pointer to the ARMulator state.

reg is the number of the coprocessor register to write to.

value is the address of the data to write to the coprocessor register.

Return

The function should return:

- `ARMul_DONE`, if the register can be written
- `ARMul_CANT`, if the register cannot be written.

12.9.13 ARMul_SetConfig

This function changes the config value of the modeled processor. The config value represents the state of the configuration pins on the ARM core. Refer to *Configuration bits and signals* on page 12-62 for details of bit to signal assignments.

Syntax

```
ARMword ARMul_SetConfig(ARMul_State *state, ARMword changed,
                                ARMword config)
```

where:

`state` is a pointer to the ARMulator state.

`changed` is a bitmask of the config bits to change.

`config` contains the new values of the bits to change.

Return

The function returns the previous config value.

Usage

Note

If a bit is cleared in `changed` it must not be set in `config`. For example, to set bit 1 and clear bit 0:

`changed`	0x03 (00000011 binary)
`config`	0x02 (00000010 binary)

`ConfigChangeUpcall()` is called. See *ConfigChangeUpcall* on page 12-62 for more information on this upcall.

Example

```
oldConfig = ARMul_SetConfig(state, 0x00000001, 0x00000001);
// This sets bit 0 to value 1
oldConfig = ARMul_SetConfig(state, 0x00000002, 0x00000001);
// This sets bit 1 to value 0 - note that bit 0 is unaffected.
```

The following call can be used to obtain the current settings of the configuration pins, without modifying them:

```
currentConfig = ARMul_SetConfig(state, 0, 0);
```

12.10 Exceptions

The following functions enable a model to set or clear interrupts and resets, or branch to a SWI handler:

- *ARMul_SetNirq and ARMul_SetNfiq* on page 12-55
- *ARMul_SetNreset* on page 12-56
- *ARMul_SWIHandler* on page 12-56.

12.10.1 ARMul_SetNirq and ARMul_SetNfiq

The following functions are used to set and clear IRQ and FIQ interrupts.

Syntax

```
unsigned ARMul_SetNirq(ARMul_State *state, unsigned value)
unsigned ARMul_SetNfiq(ARMul_State *state, unsigned value)
```

where:

`state` is a pointer to the ARMulator state.

`value` is the new `Nirq` or `Nfiq` signal value.

Note

The signals are active LOW:

- 0 = interrupt
- 1 = no interrupt.

Return

The functions return the old signal value.

12.10.2 ARMul_SetNreset

This function sets and clears RESET exceptions.

Syntax

```
unsigned ARMul_SetNreset(ARMul_State *state, unsigned value)
```

where:

state is a pointer to the ARMulator state.

value is the new `Nreset` signal value.

Note

The signal is active LOW:

- 0 = reset
- 1 = no reset.

Return

The function returns the old signal value.

12.10.3 ARMul_SWIHandler

This function can be called from a `handle_swi()` function to enter a SWI handler at a given address. It causes the processor to act as if it had taken the SWI vector, decoded the SWI number, and then branched to this address.

Syntax

```
void ARMul_SWIHandler(ARMul_State *state, ARMword address)
```

where:

state is a pointer to the ARMulator state.

address is the address of the instruction to return to.

Usage

See the code for handling `SWI_GenerateError` in `angel.c`. for an example of how to use this function.

12.11 Upcalls

The ARMulator can be made to call back your model when some state values change. You do this by installing the relevant *upcall*. In the context of the ARMulator, the term upcall is synonymous with callback.

You must provide implementations of the upcalls if you want to use them in your own models. See the implementations in the ARM supplied models for examples.

You can use upcalls to avoid having to check state values on every access. For example, a memory model is expected to present the ARM core with data in the correct endianness for the value of the ARM processor **bigend** signal. A memory model can attach to the `ConfigChangeUpcall()` upcall to be informed when this signal changes.

Every upcall is called when the ARMulator resets and after ARMulator initialization is complete, regardless of whether the signals have changed, with the exception of `UnkRDIInfoUpcall()`.

The upcalls are defined in `armdefs.h`. The following upcalls are described in the sections below:

- *ExitUpcall* on page 12-59
- *ModeChangeUpcall* on page 12-60
- *TransChangeUpcall* on page 12-61
- *ConfigChangeUpcall* on page 12-62
- *InterruptUpcall* on page 12-64
- *ExceptionUpcall* on page 12-65
- *UnkRDIInfoUpcall* on page 12-66.

Refer to *Installing and removing upcalls* on page 12-58 for information on how to install and remove the upcalls.

12.11.1 Installing and removing upcalls

Each upcall has its own installation and removal functions. These functions follow a standard format, as described below.

Installing an upcall

Each upcall in installed using a function of the form:

```
void *ARMul_Install<UpcallName>(ARMul_State *state,
                                typename *fn,
                                void *handle)
```

where:

<UpcallName>

is the name of the upcall. For example, the `ExitUpcall()` is installed with `ARMul_InstallExitHandler`. The upcalls are:

- `ExitUpcall()`
- `ModeChangeUpcall()`
- `TransChangeUpcall()`
- `ConfigChangeUpcall()`
- `InterruptUpcall()`
- `ExceptionUpcall()`
- `UnkRDIInfoUpcall()`.

state is a pointer to the ARMulator state.

typename is the type of the function, as defined by **typedef** in the upcall prototype.

fn is a pointer to the function to be installed.

handle is the handle to be passed to the corresponding Remove upcall function.

The function returns a void * handle to the upcall handler. This should be kept because it is required by the corresponding Remove upcall function.

Removing an upcall

Each upcall is removed using a function of the form:

```
int *ARMul_Remove<UpcallName>(ARMulState *state, void *node)
```

where:

`<UpcallName>`
is the name of the upcall to be removed. For example, the `ExitUpcall()` is removed with `ARMul_RemoveExitHandler`. See the description of the Install upcall function for a list of upcall names.

`state` is the state pointer.

`node` is the handle returned from the corresponding Install upcall function.

The remove upcall functions return:

- `TRUE` if the upcall is removed
- `FALSE` if the upcall remove failed.

12.11.2 ExitUpcall

The exit upcall is called when the ARMulator exits. It should be used to release any memory used.

Syntax

```
typedef void armul_ExitUpcall(void *handle)
```

where:

`handle` is a pointer to the ARMulator state.

Usage

Note

The ANSI `free()` function is a valid `ExitUpcall()`. If no exit upcall is registered and a model uses some memory, that memory will be lost.

Install the upcall using:

```
void *ARMul_InstallExitHandler(ARMul_State *state,
                               armul_ExitUpcall *fn,
                               void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveExitHandler(ARMul_State *state, void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.3 ModeChangeUpcall

The mode change upcall is called whenever the ARMulator changes mode. The upcall is passed both the *old* and *new* modes.

Syntax

```
typedef void armul_ModeChangeUpcall(void *handle, ARMword old,
                                                  ARMword new)
```

where:

handle is a pointer to the ARMulator state.

old is the old processor mode. Values for mode are defined in `armdefs.h` (see Table 12-1 on page 12-43).

new is the new processor mode. Values for mode are defined in `armdefs.h` (see Table 12-1 on page 12-43).

Usage

Install the mode change upcall using:

```
void *ARMul_InstallModeChangeHandler(ARMul_State *state,
                                     armul_ModeChangeUpcall *fn,
                                     void *handle)
```

Remove the mode change upcall using:

```
int ARMul_RemoveModeChangeHandler(ARMul_State *state,
                                  void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.4 TransChangeUpcall

This upcall is called when the **nTRANS** signal on the ARM core changes.

The **nTRANS** signal is the Not Memory Translate signal. When LOW, it indicates that the processor is in user mode, or that the processor is executing an `LDRT`/`STRT` instruction from a non-user mode. It can be used to tell memory management models when translation of the addresses should be turned on, or as an indicator of non-user mode activity (for example, to provide different levels of access in non-user modes).

Refer to the *ARM Architectural Reference Manual* for details of the `LDRT`/`STRT` instructions.

Syntax

```
typedef void armul_TransChangeUpcall(void *handle, unsigned old,
                                     unsigned new)
```

where:

handle is a pointer to the ARMulator state.

old is the old **nTRANS** signal value.

new is the new **nTRANS** signal value.

Usage

Install the upcall using:

```
void *ARMul_InstallTransChangeHandler(ARMul_State *state,
                                      armul_TransChangeUpcall *fn,
                                      void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveTransChangeHandler(ARMul_State *state,
                                   void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.5 ConfigChangeUpcall

This upcall is made when the ARMulator model configuration is changed (for example, from big-endian to little-endian). You can call `ARMul_SetConfig()` to change the configuration yourself (see *ARMul_SetConfig* on page 12-54).

Configuration is specified as a bitfield of config bits. The config bits represent signals to the configuration pins on the ARM core. Table 12-2 lists the bits that correspond to each signal.

Refer to the *ARM Architectural Reference Manual* for more information on configuration signals.

If you have a CP15 then the control register bits corresponding to the signals listed in Table 12-2 will be set in the same way.

Table 12-2 Configuration bits and signals

Signal	Bit	Notes
ARMul_Prog32	bit 4	Always high on ARM7TDMI, ARM9TDMI
ARMul_Data32	bit 5	Always high on ARM7TDMI, ARM9TDMI
ARMul_LateAbt	bit 6	Not on ARM7, ARM9, or StrongARM
ARMul_BigEnd	bit 7	
ARMul_BranchPredict	bit 11	ARM8 only

Syntax

```
typedef void armul_ConfigChangeUpcall(void *handle, ARMword old,
                                      ARMword new)
```

where:

handle is a pointer to the ARMulator state.

old is a bitfield representing the old configuration.

new is a bitfield representing the new configuration.

Usage

Install the upcall using:

```
void *ARMul_InstallConfigChangeHandler(ARMul_State *state,
                                       armul_ConfigChangeUpcall *fn,
                                       void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveConfigChangeHandler(ARMul_State *state,
                                    void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.6 InterruptUpcall

This upcall is called whenever the ARM core *notices* an interrupt (not when it takes an interrupt) or reset. It is called even if interrupts are disabled.

Syntax

```
typedef unsigned int armul_InterruptUpcall(void *handle,
                                           unsigned int which)
```

where:

handle is a pointer to the ARMulator state.

which is a bitfield that encodes which interrupt(s) have been noticed:

`bit 0`	Fast interrupt (FIQ).
`bit 1`	Interrupt request (IRQ).
`bit 2`	Reset.

Usage

This upcall can be used by a memory model to reset its state or implement a wake-up, for example. It is called at the start of the instruction or cycle (depending on the core being emulated) when the interrupt is noticed.

The interrupt responsible can be removed using `ARMul_SetNirq()` or `ARMul_SetNfiq()`, in which case the ARM will not notice the interrupt. See *ARMul_SetNirq and ARMul_SetNfiq* on page 12-55 for more information.

Note

You can use `ARMul_SetNirq()` and `ARMul_SetNfiq()` to clear the interrupt signal, but they will *not* necessarily clear the interrupt source itself.

Install the interrupt upcall using:

```
void *ARMul_InstallInterruptHandler(ARMul_State *state,
                        armul_InterruptUpcall *fn, void *handle)
```

Remove the interrupt upcall using:

```
int ARMul_RemoveInterruptHandler(ARMul_State *state, void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.7 ExceptionUpcall

This upcall is called whenever the ARM processor takes an exception.

Syntax

```
typedef unsigned int armul_ExceptionUpcall(void *handle,
                                           ARMword vector,
                                           ARMword pc,
                                           ARMword instr)
```

where:

handle is a pointer to the ARMulator state.

vector is the address of the appropriate hardware vector to be taken for the exception.

pc is the value of pc at the time the exception occurs.

instr is the instruction that caused the exception.

Usage

As an example, this can be used by an operating system model to intercept and emulate SWIs. If an installed upcall returns non-zero, the ARM does not take the exception (the exception is essentially ignored).

Note

In this release of the ARMulator, this occurs in addition to the calling of the installed operating system model's `handle_swi()` function. Future releases may not support the operating system interface, and you should use this upcall in preference. The model can be installed as a basic model (see *Basic model interface* on page 12-7). The sample models, such as `angel.c` and `validate.c`, shipped with this release of the ARMulator can be built either as a Basic model or as an Operating System model.

Note

If the processor is in Thumb state, the equivalent ARM instruction will be supplied.

Install the exception upcall using:

```
void *ARMul_InstallExceptionHandler(ARMul_State *state,
                                    armul_ExceptionUpcall *fn,
                                    void *handle)
```

Remove the exception upcall using:

```
int ARMul_RemoveExceptionHandler(ARMul_State *state, void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

12.11.8 UnkRDIInfoUpcall

`UnkRDIInfoUpcall()` functions are called if the ARMulator cannot handle an `RDIInfo` request itself. They return an `RDIError` value. The `UnkRDIInfoUpcall()` function can be used by a model extending the ARMulator's RDI interface with the debugger. For example, the profiler module (in `profiler.c`) provides the `RDIProfile` Info calls.

Syntax

```
typedef int armul_UnkRDIInfoUpcall(void *handle, unsigned type
                                   ARMword *arg1,
                                   ARMword *arg2)
```

where:

handle is a pointer to the ARMulator state.

type is the `RDI_Info` subcode. These are defined in `rdi_info.h`. See the usage section below for some examples.

arg1/arg2 are arguments passed to the upcall from the calling function.

Usage

The ARMulator stops calling `UnkRDIInfoUpcall()` functions when one returns a value other than `RDIError_UnimplementedMessage`.

The following codes are examples of the `RDI_Info` subcodes that can be specified as *type*:

`RDIInfo_Target`

This enables models to declare how to extend the functionality of the target. For example, `profiler.c` intercepts this call to set the `RDITarget_CanProfile` flag.

`RDIInfo_Points`

`watchpnt.c` intercepts `RDIInfo_Points` to tell the debugger that the ARMulator supports watchpoints. This is similar to the use of `RDIInfo_Target` in `profiler.c`.

RDIInfo_SetLog

This is passed around so that models can switch logging information on and off. For example, tracer.c uses this call to switch tracing on and off from bit 4 of the rdi_log value.

RDIRequestCyclesDesc

This enables models to extend the list of counters provided by the debugger in $statistics. Models call ARMul_AddCounterDesc() (see *ARMulator specific functions* on page 12-75) to declare each counter in turn. It is essential that the model also trap the RDICycles RDI info call.

RDICycles Models that have declared a statistics counter by trapping RDIRequestCyclesDesc (above) must also respond to RDICycles by calling ARMul_AddCounterValue() (see *ARMulator specific functions* on page 12-75) for each counter in turn, in the same order as they were declared.

The above RDI Info calls have already been dealt with by the ARMulator, and are passed for information only, or so that models can add information to the reply. Models should always respond with RDIError_UnimplementedMessage, so that the message is passed on even if the model has responded.

Install the upcalls using:

```
void *ARMul_InstallUnkRDIInfoHandler(ARMul_State *state,
                    armul_UnkRDIInfoUpcall *proc, void *handle)
```

Remove the upcalls using:

```
int ARMul_RemoveUnkRDIInfoHandler(ARMul_State *state,
                                 void *node)
```

Refer to *Installing and removing upcalls* on page 12-58 for more information.

Example

The angel.c model supplied with the ARMulator uses the UnkRDIInfoUpcall() to interact with the debugger:

RDIErrorP returns errors raised by the program running under the ARMulator to the debugger.

RDISet_Cmdline finds the command line set for the program by the debugger.

RDIVector_Catch intercepts the hardware vectors.

12.12 Memory access functions

The memory model can be probed by another model using a set of functions for reading and writing memory. These functions access memory without inserting cycles on the bus. If your model needs to insert cycles on the bus, it should install itself as a memory model, possibly between the core and the real memory model.

12.12.1 Reading from a given address

The following functions return the word, halfword, or byte at the specified address. Each function accesses the memory without inserting cycles on the bus.

Syntax

```
ARMword ARMul_ReadWord(ARMul_State *state, ARMword address)
ARMword ARMul_ReadHalfWord(ARMul_State *state, ARMword address)
ARMword ARMul_ReadByte(ARMul_State *state, ARMword address)
```

where:

state is a pointer to the ARMulator state.

address is the address in emulated memory from which the word, halfword, or byte is to be read.

Return

The functions return the word, halfword, or byte, as appropriate.

12.12.2 Writing to a specified address

The following functions write the specified word, halfword, or byte at the specified address. Each function accesses memory without inserting cycles on the bus.

Syntax

```
void ARMul_WriteWord(ARMul_State *state, ARMword address,
                     ARMword data)
```

```
void ARMul_WriteHalfWord(ARMul_State *state, ARMword address,
                         ARMword data)
```

```
void ARMul_WriteByte(ARMul_State *state, ARMword address,
                     ARMword data)
```

where:

state is a pointer to the ARMulator state.

address is the address in emulated memory to write to.

data is the word, halfword, or byte to write.

Note

It is not possible to tell if these calls resulted in a data abort.

12.13 Event scheduling functions

The event scheduling functions enable you to schedule a call to a function based on:

- the number of instructions executed (*instruction events*)
- the number of memory system cycles (*cycle events*).

This section describes the event scheduling functions:

Instruction events The following functions enable you to schedule instruction events:

- *armul_Hourglass* on page 12-71
- *ARMul_HourglassSetRate* on page 12-72.

Cycle events The following functions enable you to schedule cycle events:

- *ARMul_ScheduleEvent* on page 12-73
- *ARMul_ScheduleCoreEvent* on page 12-74.

For details on ARMulator events see *Events* on page 12-87.

12.13.1 armul_Hourglass

The `armul_Hourglass()` function provides a mechanism for calling a function at every instruction, or at every `n` instructions for a value of `n` that is set by the `ARMul_HourglassSetRate()` function.

Syntax

```
typedef void armul_Hourglass(void *handle, ARMword pc,
                             ARMword instr)
```

where:

`handle` is a pointer to the ARMulator state.

`pc` is the program counter.

`instr` is the instruction about to be executed.

Usage

Install the function in the same way as upcalls:

```
void *ARMul_InstallHourglass(ARMul_State *state,
                             armul_Hourglass *fn,
                             void *handle)
```

Remove function with:

```
int ARMul_RemoveHourglass(ARMul_State *state, void *node)
```

The remove function returns:

- `TRUE`, if the hourglass function is removed successfully
- `FALSE`, if the hourglass function is not removed successfully.

You can use the `ARMul_HourglassSetRate()` function to change the default rate at which `armul_Hourglass()` is called. See below for details.

12.13.2 ARMul_HourglassSetRate

This function sets the rate at which the `armul_HourGlass()` function is called. By default, the `armul_HourGlass()` function is called every instruction.

Syntax

```
unsigned long ARMul_HourglassSetRate(ARMul_State *state,
                                     void *node,
                                      unsigned long rate)
```

where:

state is a pointer to the ARMulator state.

rate defines the rate at which the function should be called. For example, a value of 1 calls the function every instruction. A value of 100 calls it every 100 instructions.

node is the handle returned from `ARMul_InstallHourglass()` when the upcall was installed.

Return

The function returns the old hourglass rate.

12.13.3 ARMul_ScheduleEvent

This function schedules events using memory system cycles. It enables a function to be called at a specified number of cycles in the future.

Syntax

```
void ARMul_ScheduleEvent(ARMul_State *state,
                         unsigned long delay,
                         armul_EventProc *func,
                         void *handle)
```

where:

state is a pointer to the ARMulator state.

func is a pointer to the event function to call of type:

```
typedef unsigned armul_EventProc(void *handle)
```

delay specifies the number of cycles to delay before the event function is called.

handle is the void * handle to call the event function with.

Note

The function can be called only on the first instruction boundary following the specified cycle.

12.13.4 ARMul_ScheduleCoreEvent

`ARMmul_ScheduleCoreEvent()` function schedules events using the absolute core cycle count at which the event will occur. It enables a function to be called at a specified point in the future.

Syntax

```
void ARMul_ScheduleEventCore(ARMul_State *state,
                             armul_EventProc *func,
                             void *handle,
                             unsigned long coreCycleCount)
```

where:

state is a pointer to the ARMulator state.

func is a pointer to the event function to call, of type:

typedef unsigned armul_EventProc(**void** **handle*)

handle is the void *handle to call the event function with.

coreCycleCount the absolute core cycle count at which the event function is to be called.

Note

This function is supported only by ARM9-based models.

12.14 ARMulator specific functions

The following are general purpose ARMulator functions. They include functions to access processor properties, add counter descriptions and values, stop the ARMulator and execute code:

- *ARMul_RaiseError* on page 12-75
- *ARMul_Time* on page 12-77
- *ARMul_Properties* on page 12-77
- *ARMul_CondCheckInstr* on page 12-78
- *ARMul_AddCounterDesc* on page 12-79
- *ARMul_AddCounterValue* on page 12-80
- *ARMul_HaltEmulation* on page 12-81
- *ARMul_EndCondition* on page 12-81
- *ARMul_DoProg* on page 12-82
- *ARMul_DoInstr* on page 12-82.

12.14.1 ARMul_RaiseError

Errors of type `ARMul_Error` are returned from a number of initialization and installation functions. These errors should be passed through `ARMul_RaiseError()`. This is a printf-like function that formats the error message associated with an `ARMul_Error` error code.

Syntax

```
ARMul_Error ARMul_RaiseError(ARMul_State *state,
                             ARMul_Error errcode, ...)
```

where:

`state` is a pointer to the ARMulator state.

`errcode` is the error code for the error message to be formatted.

`...` are printf-style format specifiers of variadic type.

Return

The function returns the error code it was passed, after formatting the error message.

Example

This function is a printf-style variadic function, and the textual form can be a printf-style format string. For example:

```
interf->handle = (model_state *)malloc(sizeof(model_state));
if (interf->handle == NULL)
       return ARMul_RaiseError(state, ARMulErr_OutOfMemory);
```

For example, the `ARMulErr_MemTypeUnhandled` error message, used by memory models to reject an unrecognized interface type, is declared:

```
ERROR(ARMulErr_MemTypeUnhandled,
       "Memory model ’%s’ incompatible with bus interface.")
```

and called:

```
return ARMul_RaiseError(state,
                        ARMulErr_MemTypeUnhandled,
                        ModelName);
```

In this case, the debugger displays an error message such as:

```
Memory model ’Flat’ incompatible with bus interface.
```

Extending the error file

The file `errors.h` can be extended by adding more errors. However, new errors can only be added at the *end* of the file.

Entries are of the form:

```
ERROR(ARMulErr_OutOfMemory, "Out of memory.")
```

This declares an error message, `ARMulErr_OutOfMemory`, with the textual form:

```
       "Out of memory"
```

12.14.2 ARMul_Time

This function returns the number of memory cycles executed since system reset.

Syntax

`unsigned long` `ARMul_Time(ARMul_State *`*`state`*`)`

where:

`state` is a pointer to the ARMulator state.

Return

The function returns the total number of cycles executed since system reset.

12.14.3 ARMul_Properties

This function returns the properties word associated with the processor being emulated.

This is a bitfield of properties, defined in `armdefs.h`.

Syntax

`ARMword ARMul_Properties(ARMul_State *`*`state`*`)`

where:

`state` is a pointer to the ARMulator state.

Return

The function returns the properties word. This is a bitfield of properties, defined in `armdefs.h`.

12.14.4 ARMul_CondCheckInstr

Given an instruction, the `ARMul_CondCheckInstr()` function returns `TRUE` if it would execute given the current state of the PSR flags.

Syntax

unsigned `ARMul_CondCheckInstr(ARMul_State *`*`state`*`, ARMword `*`instr`*`)`

where:

`state` is a pointer to the ARMulator state.

`instr` is the instruction opcode to check.

Return

The function returns:

- `TRUE` if the instruction would execute
- `FALSE` if the instruction would not execute.

12.14.5 ARMul_AddCounterDesc

The `ARMul_AddCounterDesc()` function adds new counters to `$statistics`.

Syntax

```
int ARMul_AddCounterDesc(ARMul_State *state,
                         ARMword *arg1,
                         ARMword *arg2,
                         const char *name)
```

where:

state is a pointer to the ARMulator state.

arg1/arg2 are the arguments passed to the `UnkRDIInfoUpcall()` .

name is a string that names the statistic counter. The string must be less than 32 characters long.

Return

The function returns one of:

- `RDIError_BufferFull`
- `RDIError_UnimplementedMessage.`

Usage

When the ARMulator receives an `RDIRequestCycleDesc()` call from the Debugger, it uses the `UnkRDIInfoUpcall()` (see *Upcalls* on page 12-57) to ask each module in turn if it wishes to provide any statistics counters. Each module responds by calling `ARMul_AddCounterDesc()` with the arguments passed to the `UnkRDIInfoUpcall()`.

All statistics counters must be either a 32-bit or 64-bit word, and be monotonically increasing. That is, the statistic value must go up over time. This is a requirement because of the way the Debugger calculates `$statistics_inc`.

See the implementation in `armflat.c` for an example.

12.14.6 ARMul_AddCounterValue

This function is called when the debugger requests the current statistics values.

Syntax

```
int ARMul_AddCounterValue(ARMul_State *state,
                          ARMword *arg1,
                          ARMword *arg2,
                          bool is64,
                          const ARMword *counter)
```

where:

state is a pointer to the ARMulator state. .

arg1/arg2 are the arguments passed to the `UnkRDIInfoUpcall()`.

is64 denotes whether the counter is a pair of 32-bit words making a 64-bit counter (least significant word first), or a single 32-bit value. This enables modules to provide a full 64-bit counter.

counter is the current value of the counter.

Return

The function must always return `RDIError_UnimplementedMessage`.

Usage

When the ARMulator receives an `RDICycles()` call from the debugger, it uses the `UnkRDIInfoUpcall()` to ask each module in turn to provide the counter values. Each module responds by calling `ARMul_AddCounterValue()`.

——— **Note** ———

It is essential that a module that calls `ARMul_AddCounterDesc()` when `RDIRequestCycleDesc()` is called also calls `ARMul_AddCounterValue()` when `RDICycles()` is called. It must also call both functions the same number of times and in the same order.

12.14.7 ARMul_HaltEmulation

This function stops emulator execution at the end of the current instruction, giving a reason code.

Syntax

```
void ARMul_HaltEmulation(ARMul_State *state,
                         unsigned end_condition)
```

where:

state is a pointer to the ARMulator state.

end_condition

is one of the `RDIError` error values defined in `rdi_err.h`. Not all of these errors are valid. The debugger interprets *end_condition* and issues a suitable message.

12.14.8 ARMul_EndCondition

This function returns the *end_condition* passed to `ARMul_HaltEmulation()`.

Syntax

```
unsigned ARMul_EndCondition(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The end condition passed to `ARMul_HaltEmulation()`.

12.14.9 ARMul_DoProg

This function starts running the emulator at the current pc value. It is called from the ARMulator RDI interface.

Syntax

```
ARMword ARMul_DoProg(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the value of pc on halting emulation.

12.14.10 ARMul_DoInstr

This function executes a single instruction. It is called from the ARMulator RDI interface.

Syntax

```
ARMword ARMul_DoInstr(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the value of pc on halting emulation.

12.15 Accessing the debugger

This section describes the input, output, and RDI functions that you can use to access the debugger. There are two groups of debugger functions:

Input/output functions

Several functions are provided to display messages in the host debugger. Under armsd these functions print messages to the console. Under ADW/ADU they display messages to the relevant window:

- *ARMul_DebugPrint* on page 12-83 displays a message on the RDI console window
- *ARMul_ConsolePrint* on page 12-84 displays a message on the debugger console window
- *ARMul_PrettyPrint* on page 12-84 displays a formatted message.
- *ARMul_DebugPause* on page 12-85 waits for user input before continuing the message display.

RDI functions

These are:

- *ARMul_RDILog* on page 12-85
- *ARMul_HostIf* on page 12-86.

12.15.1 ARMul_DebugPrint

This function displays a message in the RDI logging window under ADW/ADU, or to the console under armsd.

Syntax

```
void ARMul_DebugPrint(ARMul_State *state, const char *format,
                      ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

12.15.2 ARMul_ConsolePrint

This function prints the text specified in the format string to the ARMulator console. Under ADW/ADU, the text appears in the console window.

Syntax

```
void ARMul_ConsolePrint(ARMul_State *state, const char *format,
                        ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

Note

Use `ARMul_PrettyPrint()` to display startup messages.

12.15.3 ARMul_PrettyPrint

This function prints a string in the same way as `ARMul_ConsolePrint()`, but in addition performs line break checks so that word wrap is avoided. It should be used for displaying startup messages.

Syntax

```
void ARMul_PrettyPrint(ARMul_State *state, const char *format,
                       ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

12.15.4 ARMul_DebugPause

This function waits for the user to press any key.

Syntax

```
void ARMul_DebugPause(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

12.15.5 ARMul_RDILog

This function returns the value of the RDI logging level.

Syntax

```
ARMword ARMul_RDILog(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

12.15.6 ARMul_HostIf

This function returns a pointer to a `RDI_HostosInterface` structure, defined in `rdi_hif.h`. The structure includes pointers to RDI functions that enable a debug target to send and received textual information to and from a host.

Syntax

```
const RDI_HostosInterface *ARMul_HostIf(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns a pointer to the `RDI_HostosInterface` structure. Refer to `rdi_hif.h` for the `RDI_HostosInterface` structure definition.

Usage

An operating system model can make use of this to:

- efficiently access the console window (under ADW/ADU) or the console (under armsd) without going through `ARMul_ConsolePrint()`
- receive user input.

The following input/output functions are included in `RDI_HostosInterface`:

```
void writec(RDI_Hif_HostosArg *arg, int c)
```

Writes a single character to the console window under ADW/ADU, or to the console under armsd. This is used by `ARMul_ConsolePrint()`, and by the emulation of `SYS_WriteC` in `angel.c`.

```
int readc(RDI_Hif_HostosArg *arg, char const *buffer, int len)
```

Reads a single character of input from the host debugger.

```
int write(RDI_Hif_HostosArg *arg, char const *buffer, int len)
```

Writes a stream of data to the console window under ADW/ADU, or to the console under armsd.

```
char *gets(RDI_Hif_HostosArg *arg, char *buffer, int len)
```

Reads a string from the host debugger.

12.16 Events

The ARMulator has a mechanism for broadcasting and handling events. These events consist of an event number and a pair of words. The number identifies the event. The semantics of the words depends on the event.

The core ARMulator generates some example events, defined in `armdefs.h`. They are divided into three groups:

- events from ARM processor core in Table 12-3
- events from MMU and cache (not on StrongARM-110) in Table 12-4 on page 12-88
- events from prefetch unit (ARM8-based processors only) in Table 12-5 on page 12-88.

These events can be logged in the trace file if tracing is enabled, and trace events is turned on. Additional modules can provide new event types that will be handled in the same way.

You can catch events by installing an event upcall (see *armul_EventUpcall* on page 12-89). You can raise an event by calling `ARMul_RaiseEvent()` (see *ARMul_RaiseEvent* on page 12-90).

Refer to Chapter 12 *ARMulator* in the *ARM Software Development Toolkit User Guide* for more information and examples.

Table 12-3 Events from ARM processor core

Event name	Word 1	Word 2	Event number
`CoreEvent_Reset`	-	-	0x1
`CoreEvent_UndefinedInstr`	pc value	instruction	0x2
`CoreEvent_SWI`	pc value	SWI number	0x3
`CoreEvent_PrefetchAbort`	pc value	-	0x4
`CoreEvent_DataAbort`	pc value	aborting address	0x5
`CoreEvent_AddrExceptn`	pc value	aborting address	0x6
`CoreEvent_IRQ`	pc value	-	0x7
`CoreEvent_FIQ`	pc value	-	0x8
`CoreEvent_Breakpoint`	pc value	RDI_PointHandle	0x9
`CoreEvent_Watchpoint`	pc value	Watch address	0xa

Table 12-3 Events from ARM processor core (Continued)

Event name	Word 1	Word 2	Event number
`CoreEvent_IRQSpotted`	pc value	-	0x17
`CoreEvent_FIQSpotted`	pc value	-	0x18
`CoreEvent_ModeChange`	pc value	new mode	0x19
`CoreEvent_Dependency`	pc value	interlock register bitmask	0x20

Table 12-4 Events from MMU and cache (not on StrongARM-110)

Event name	Word 1	Word 2	Event number
`MMUEvent_DLineFetch`	miss address	victim address	0x10001
`MMUEvent_ILineFetch`	miss address	victim address	0x10002
`MMUEvent_WBStall`	physical address of write	number of words in write buffer	0x10003
`MMUEvent_DTLBWalk`	miss address	victim address	0x10004
`MMUEvent_ITLBWalk`	miss address	victim address	0x10005
`MMUEvent_LineWB`	miss address	victim address	0x10006
`MMUEvent_DCacheStall`	address causing stall	address fetching	0x10007
`MMUEvent_ICacheStall`	address causing stall	address fetching	0x10008

Table 12-5 Events from prefetch unit (ARM810 only)

Event name	Word 1	Word 2	Event number
`PUEvent_Full`	next pc value	-	0x20001
`PUEvent_Mispredict`	address of branch	-	0x20002
`PUEvent_Empty`	next pc value	-	0x20003

12.16.1 armul_EventUpcall

This upcall catches ARMulator events.

Syntax

```
typedef void armul_EventUpcall(void *handle, unsigned int event,
                               ARMword addr1, ARMword addr2)
```

where:

handle is a pointer to the ARMulator state.

event is one of the event numbers defined in Table 12-3, Table 12-3, and Table 12-3 on page 12-87.

addr1 is the first word of the event (see the Tables above).

addr2 is the second word of the event (see the Tables above).

Usage

Install the upcall using:

```
void *ARMul_InstallEventUpcall(ARMul_State *state,
                               armul_EventUpcall *fn,
                               void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveEventUpcall(ARMul_State *state, void *node)
```

12.16.2 ARMul_RaiseEvent

This function invokes events. The events are passed to the user-supplied event upcalls.

Syntax

```
void ARMul_RaiseEvent(ARMul_State *state, unsigned int event,
                      ARMword word1, ARMword word2)
```

where:

handle is a pointer to the ARMulator state.

event is one of the event numbers defined in Table 12-3, Table 12-3, and Table 12-3 on page 12-87.

addr1 is the first word of the event (see the Tables above).

addr2 is the second word of the event (see the Tables above).

Chapter 13
ARM Image Format

This chapter describes the ARM Image Format (AIF). It contains the following sections:

13.1 Overview of the ARM Image Format

ARM Image Format (AIF) is a simple format for ARM executable images, consisting of:

- a 128-byte header
- the image code
- the image initialized static data.

An AIF image is capable of self-relocation if it is created with the appropriate linker options. The image can be loaded anywhere and it will execute where it is loaded. After an AIF image has been relocated, it can create its own zero-initialized area. Finally, the image is entered at the unique entry point.

13.2 AIF variants

There are three variants of AIF:

Executable AIF

Executable AIF can be loaded at its load address and entered at the same point (at the first word of the AIF header). It prepares itself for execution by relocating itself if required and setting to zero its own zero-initialized data.

The header is part of the image itself. Code in the header ensures that the image is properly prepared for execution before being entered at its entry address.

The fourth word of an executable AIF header is:

`BL` *`entrypoint`*

The most significant byte of this word (in the target byte order) is 0xeb.

The base address of an executable AIF image is the address at which its header should be loaded. Its code starts at *base* + 0x80.

Non-executable AIF

Non-executable AIF must be processed by an image loader that loads the image at its load address and prepares it for execution as detailed in the AIF header. The header is then discarded. The header is not part of the image, it only describes the image.

The fourth word of a non-executable AIF image is the offset of its entry point from its base address. The most significant nibble of this word (in the target byte order) is 0x0.

The base address of a non-executable AIF image is the address at which its code should be loaded.

Extended AIF

Extended AIF is a special type of non-executable AIF. It contains a scatter-loaded image. It has an AIF header that points to a chain of load region descriptors within the file. The image loader should place each region at the location in memory specified by the load region descriptor.

13.3 The layout of AIF

This section describes the layout of AIF images.

13.3.1 AIF image layout

An AIF image has the following layout:

- Header
- Read-only area
- Read-write area
- Debugging data (optional)
- Self-relocation code (position-independent)
- Relocation list. This is a list of byte offsets from the beginning of the AIF header, of words to be relocated, followed by a word containing `-1`. The relocation of non-word values is not supported.

Note

An AIF image is restartable if, and only if, the program it contains is restartable (an AIF image is *not* reentrant). Following self-relocation, the second word of the header must be reset to `NOP`. This causes no additional problems with the read-only nature of the code section.

On systems with memory protection, the self-relocation code must be bracketed by system calls to change the access status of the read-only section (first to writable, then back to read-only).

13.3.2 Debugging data

After the execution of the self-relocation code, or if the image is not self-relocating, the image has the following layout:

- Header
- Read-only area
- Read-write area
- Debugging data (optional).

AIF images support being debugged by an ARM debugger. Low-level and source-level support are orthogonal. An AIF image can have both, either, or neither kind of debugging support.

References from debugging tables to code and data are in the form of relocatable addresses. After loading an image at its load address these values are effectively absolute.

References between debugger table entries are in the form of offsets from the beginning of the debugging data area. Following relocation of a whole image, the debugging data area itself is position-independent and may be copied or moved by the debugger.

13.3.3 AIF header

Table 13-1 shows the layout of the AIF header.

Table 13-1 AIF header layout

00:	`NOP`[a]	
04:	`BL` SelfRelocCode	NOP if the image is not self-relocating
08:	`BL` ZeroInit	NOP if the image has none.
0C:	`BL` ImageEntryPoint or EntryPoint Offset	`BL` to make the header addressable via r14 ...but the application will not return... Non-executable AIF uses an offset, not `BL`. `BL` is used to make the header addressable via r14 in a position-independent manner, and to ensure that the header will be position-independent.
10:	Program Exit Instruction	… last attempt in case of return. The Program Exit Instruction is usually a `SWI` causing program termination. On systems that do not implement a SWI for this purpose, a branch-to-self is recommended. Applications are expected to exit directly and *not* to return to the AIF header, so this instruction should never be executed. The ARM linker sets this field to SWI 0x11 by default, but it may be set to any desired value by providing a template for the AIF header in an area called `AIF_HDR` in the *first* object file in the input list to armlink.
14:	Image ReadOnly size	Image ReadOnly Size includes the size of the AIF header only if the AIF type is executable (that is, if the header itself is part of the image).
18:	Image ReadWrite size	Exact size (a multiple of 4 bytes).
1C:	Image Debug size	Exact size (a multiple of 4 bytes). Includes high-level and low-level debug size. Bits 0-3 hold the type. Bits 4-31 hold the low level debug size.
20:	Image zero-init size	Exact size (a multiple of 4 bytes).

Table 13-1 AIF header layout (Continued)

24:	Image debug type	Valid values for Image debug type are: **0** No debugging data present. **1** Low-level debugging data present. **2** Source level debugging data present. **3** 1 and 2 are present together. All other values of image debug type are reserved.
28:	Image base	Address where the image (code) was linked.
2C:	Work space	Obsolete.
30:	Address mode: 26/32 + 3 flag bytes	The word at offset 0x30 is 0, or contains in its least significant byte (using the byte order appropriate to the target): **26** Indicates that the image was linked for a 26-bit ARM mode, and may not execute correctly in a 32-bit mode. This is obsolete. **32** Indicates that the image was linked for a 32-bit ARM mode, and may not execute correctly in a 26-bit mode. A value of 0 indicates an old-style 26-bit AIF header. If the Address mode word has bit 8 set, the image was linked with separate code and data bases (usually the data is placed immediately after the code). The word at offset 0x34 contains the base address of the image's data.
34:	Data base	Address where the image data was linked.
38:	Two reserved words (initially 0)	In Extended AIF images, the word at 0x38 is non-zero. It contains the byte offset within the file of the header for the first non-root load region. This header has a size of 44 bytes, and the following format: **word 0** file offset of header of next region (0 is none) **word 1** load address **word 2** size in bytes (a multiple of 4) **char[32]** the region name padded out with zeros. The initializing data for the region follows the header.
40:	`NOP`	
44:	Zero-init code 15 words as below	Header is 32 words long.

a. In all cases, `NOP` is encoded as `MOV r0,r0`

Chapter 14
ARM Object Library Format

This chapter describes the ARM Object Library Format (ALF). It contains the following sections:

14.1 Overview of ARM Object Library Format

This section defines a file format called *ARM Object Library Format* (ALF), that is used by the ARM linker and the ARM object librarian.

A library file contains a number of separate but related pieces of data. In order to simplify access to these data, and to provide for a degree of extensibility, the library file format is itself layered on another format called *Chunk File Format*. This provides a simple and efficient means of accessing and updating distinct chunks of data within a single file. Refer to *Chunk file format* on page 15-4 for a description of the Chunk File Format.

The Library format defines four chunk classes:

- Directory
- Time stamp
- Version
- Data.

There may be many *Data* chunks in a library.

The Object Library Format defines two additional chunks:

- Symbol table
- Symbol table time stamp.

14.2 Endianness and alignment

For data in a file, *address* means *offset from the start of the file*.

There is no guarantee that the endianness of an ALF file will be the same as the endianness of the system used to process it (the endianness of the file is always the same as the endianness of the target ARM system).

The two sorts of ALF cannot meaningfully be mixed (the target system cannot have mixed endianness, it must have one or the other). The ARM linker accepts inputs of either sex and produces an output of the same sex, but rejects inputs of mixed endianness.

14.2.1 Alignment

Strings and bytes may be aligned on any byte boundary.

ALF fields defined in this document do not use halfwords, and align words on 4-byte boundaries.

Within the contents of an ALF file (within the data contained in `OBJ_AREA` chunks, see below), the alignment of words and halfwords is defined by the use to which ALF is being put. For all current ARM-based systems, alignment is strict.

14.3 Library file format

For library files, the first part of each chunk name is LIB_. For object libraries, the names of the additional two chunks begin with OFL_.

Each piece of a library file is stored in a separate, identifiable chunk. Table 14-1 shows the chunk names.

Table 14-1 Library File Chunks

Chunk	Chunk name
Directory	LIB_DIRY
Time stamp	LIB_TIME
Version	LIB_VRSN
Data	LIB_DATA
Symbol table	OFL_SYMT object code
Time stamp	OFL_TIME object code

There may be many LIB_DATA chunks in a library, one for each library member. In all chunks, word values are stored with the same byte order as the target system. Strings are stored in ascending address order, which is independent of target byte order.

14.3.1 Earlier versions of ARM object library format

These notes ensure maximum robustness with respect to earlier, now obsolete, versions of the ARM object library format:

- Applications which create libraries or library members should ensure that the LIB_DIRY entries they create contain valid time stamps.
- Applications which read LIB_DIRY entries should not rely on any data beyond the end of the name string being present, unless the difference between the DataLength field and the name-string length allows for it. Even then, the contents of a time stamp should be treated cautiously.
- Applications which write LIB_DIRY or OFL_SYMT entries should ensure that padding is done with NULL (0) bytes. Applications that read LIB_DIRY or OFL_SYMT entries should make no assumptions about the values of padding bytes beyond the first, string-terminating NULL byte.

14.3.2 LIB_DIRY

The LIB_DIRY chunk contains a directory of the modules in the library, each of which is stored in a LIB_DATA chunk. The directory size is fixed when the library is created. The directory consists of a sequence of variable length entries, each an integral number of words long. The number of directory entries is determined by the size of the LIB_DIRY chunk. Table 14-2 shows the layout.

Table 14-2 The LIB_DIRY chunk

ChunkIndex	
EntryLength	The size of this LIB_DIRY chunk (an integral number of words).
DataLength	The size of the Data (an integral number of words).
Data	

where:

ChunkIndex is a word containing the zero-origin index within the chunk file header of the corresponding LIB_DATA chunk. Conventionally, the first three chunks of an OFL file are LIB_DIRY, LIB_TIME and LIB_VRSN, so ChunkIndex is at least 3. A ChunkIndex of 0 means the directory entry is unused.

The corresponding LIB_DATA chunk entry gives the offset and size of the library module in the library file.

EntryLength is a word containing the number of bytes in this LIB_DIRY entry, always a multiple of 4.

DataLength is a word containing the number of bytes used in the data section of this LIB_DIRY entry, also a multiple of 4.

Data consists of, in order:

- a zero-terminated string (the name of the library member). Strings should contain only ISO-8859 non-control characters (codes [0-31], 127 and 128+[0-31] are excluded). The string field is the name used to identify this library module. Typically it is the name of the file from which the library member was created.
- any other information relevant to the library module (often empty).

- a two-word, word-aligned time stamp. The format of the time stamp is described in *Time stamps* on page 14-7. Its value is an encoded version of the last-modified time of the file from which the library member was created.

14.3.3 LIB_VRSN

The version chunk contains a single word whose value is 1.

14.3.4 LIB_DATA

A `LIB_DATA` chunk contains one of the library members indexed by the `LIB_DIRY` chunk. The endianness or byte order of this data is, by assumption, the same as the byte order of the containing library/chunk file.

No other interpretation is placed on the contents of a member by the library management tools. A member could itself be a file in chunk file format or even another library.

14.4 Time stamps

A library time stamp is a pair of words that encode:

- a six byte count of centiseconds since 00:00:00 1st January 1900
- a two byte count of microseconds since the last centisecond.

First (most significant) word

Contains the most significant 4 bytes of the 6 byte centisecond count.

Second (least significant) word

Contains the least significant two bytes of the six byte centisecond count in the most significant half of the word and the two byte count of microseconds since the last centisecond in the least significant half of the word. This is usually 0.

Time stamp words are stored in target system byte order. They must have the same endianness as the containing chunk file.

14.4.1 LIB_TIME

The `LIB_TIME` chunk contains a two-word (eight-byte) time stamp recording when the library was last modified.

14.5 Object code libraries

An object code library is a library file whose members are files in ARM Object Format. An object code library contains two additional chunks:

- an external symbol table chunk named `OFL_SYMT`
- a time stamp chunk named `OFL_TIME.`

14.5.1 OFL_SYMT

The external symbol table contains an entry for each external symbol defined by members of the library, together with the index of the chunk containing the member defining that symbol.

The `OFL_SYMT` chunk has exactly the same format as the `LIB_DIRY` chunk except that the Data section of each entry contains only a string, the name of an external symbol, and between one and four bytes of NULL padding, as follows:

Table 14-3 OFL_SYMT chunk layout

ChunkIndex	
EntryLength	The size of this `OFL_SYMT` chunk (an integral number of words).
DataLength	The size of the External Symbol Name and Padding (an integral number of words).
External Symbol Name	
Padding	

`OFL_SYMT` entries do not contain time stamps.

14.5.2 OFL_TIME

The `OFL_TIME` chunk records when the `OFL_SYMT` chunk was last modified and has the same format as the `LIB_TIME` chunk (see *Time stamps* on page 14-7).

Chapter 15
ARM Object Format

This chapter describes the ARM Object Format. It contains the following sections:

15.1 ARM Object Format

This section describes the ARM Object Format (AOF).

The following terms apply throughout this section:

object file refers to a file in ARM Object Format.

address for data in a file, this means *offset from the start of the file.*

15.1.1 Areas

An object file written in AOF consists of any number of named, attributed areas. Attributes include:

- read-only
- reentrant
- code
- data
- position-independent.

For details see *Attributes and alignment* on page 15-9.

Typically, a compiled AOF file contains a read-only code area, and a read-write data area (a zero-initialized data area is also common, and reentrant code uses a separate based area for address constants).

15.1.2 Relocation directives

Associated with each area is a (possibly empty) list of relocation directives which describe locations that the linker will have to update when:

- a non-zero base address is assigned to the area
- a symbolic reference is resolved.

Each relocation directive may be given relative to the (not yet assigned) base address of an area in the same AOF file, or relative to a symbol in the symbol table. Each symbol may:

- have a definition within its containing object file which is local to the object file
- have a definition within the object file which is visible globally (to all object files in the link step)
- be a reference to a symbol defined in some other object file.

15.1.3 Byte sex or endianness

An AOF file can be produced in either little-endian or big-endian format.

There is no guarantee that the endianness of an AOF file will be the same as the endianness of the system used to process it (the endianness of the file is always the same as the endianness of the target ARM system).

15.1.4 Alignment

Strings and bytes may be aligned on any byte boundary. AOF fields defined in this document make no use of halfwords and align words on 4-byte boundaries.

Within the contents of an AOF file, the alignment of words and halfwords is defined by the use to which AOF is being put. For all current ARM-based systems, words are aligned on 4-byte boundaries and halfwords on 2-byte boundaries.

15.2 Overall structure of an AOF file

An AOF file contains a number of separate pieces of data. To simplify access to the data, and to give a degree of extensibility to tools which process AOF, the object file format is itself layered on another format called *Chunk File Format*, which provides a simple and efficient means of accessing and updating distinct chunks of data within a single file.

15.2.1 Chunk file format

A file written in chunk file format consists of a header, and one or more chunks. The header is always positioned at the beginning of the file. A chunk is accessed through the header. The header contains the number, size, location, and identity of each chunk in the file.

The size of the header may vary between different chunk files, but it is fixed for each file. Not all entries in a header need be used, thus limited expansion of the number of chunks is permitted without a wholesale copy. A chunk file can be copied without knowledge of the contents of its chunks.

Chunk file header

The chunk file header consists of two parts:

- the first part is a fixed length part of three words
- the second part contains a four word entry for each chunk in the file.

The first part of the header contains the following three word sized fields:

`ChunkFileId` Marks the file as a chunk file. Its value is 0xC3CBC6C5. The endianness of the chunk file can be determined from this value (if it appears to be 0xC5C6CBC3 when read as a word, each word value must be byte-reversed before use).

`max_chunks` Defines the number of the entries in the header, fixed when the file is created.

`num_chunks` Defines how many chunks are currently used in the file, which can vary from 0 to `max_chunks`. It is redundant in that it can be found by scanning the entries.

The second part of the header contains a four word entry for each chunk in the file. The number of entries is given by the `num_chunks` field in the first part of the header.

`chunkId` Is an 8-byte field identifying what data the chunk contains. Note that this is an 8-byte field, *not* a 2-word field, so it has the same byte order independent of endianness.

`file_offset` Is a one-word field defining the byte offset within the file of the start of the chunk. All chunks are word-aligned, so it must be divisible by four. A value of zero indicates that the chunk entry is unused.

`size` Is a one-word field defining the exact byte size of the chunk's contents (which need not be a multiple of four).

Identifying data types

The `chunkId` field provides a conventional way of identifying what type of data a chunk contains. It has eight characters, and is split into two parts:

- the first four characters contain a unique name allocated by a central authority
- the remaining four characters can be used to identify component chunks within this domain.

The eight characters are stored in ascending address order, as if they formed part of a NULL-terminated string, independent of endianness.

For AOF files, the first part of each chunk name is `OBJ_`. The second components are defined in the following section.

15.2.2 ARM object format

Each piece of an object file is stored in a separate, identifiable chunk. AOF defines five chunks as shown in Table 15-1.

Table 15-1 AOF chunks

Chunk	Chunk name
AOF Header	`OBJ_HEAD`
Areas	`OBJ_AREA`
Identification	`OBJ_IDFN`
Symbol Table	`OBJ_SYMT`
String Table	`OBJ_STRT`

Only the `AOF Header` and `AREAS` chunks must be present, but a typical object file contains all five of the above chunks.

Each name in an object file is encoded as an offset into the string table, stored in the `OBJ_STRT` chunk *The String Table Chunk (OBJ_STRT)* on page 15-21. This allows the variable-length nature of names to be factored out from primary data formats.

A feature of ARM Object Format is that chunks may appear in any order in the file (for example, the ARM C compiler and the ARM assembler produce their AOF chunks in different orders).

A language translator or other utility may add additional chunks to an object file, for example, a language-specific symbol table or language-specific debugging data. Therefore it is conventional to allow space in the chunk header for additional chunks. Space for eight chunks is conventional when the AOF file is produced by a language processor which generates all five chunks described here.

Note

The AOF header chunk should not be confused with the chunk file header.

15.3 The AOF header chunk (OBJ_HEAD)

The AOF header consists of two contiguous parts:

- the first part is a fixed size part of six words that describes the contents and nature of the object file.
- the second part has a variable length (specified in the first part of the header), and consists of a sequence of *area headers* describing the areas within the `OBJ_AREA` chunk.

Part one contains the following word sized fields:

`Object File Type`

The value 0xC5E2D080 marks the file as being in *relocatable object format* (the usual output of compilers and assemblers and the usual input to the linker). The endianness of the object code can be deduced from this value and must be identical to the endianness of the containing chunk file.

`Version Id`

Encodes the AOF version number. The current version number is 310 (0x136).

`Number of Areas`

The code and data of an object file are encapsulated in a number of separate areas in the `OBJ_AREA` chunk, each with a name and some attributes (see *Attributes and alignment* on page 15-9).

Each area is described in the variable-length part of the AOF header which immediately follows the fixed part. `Number_of_Areas` gives the number of areas in the file and, equivalently, the number of `AREA` declarations that follow the fixed part of the AOF header.

`Number of Symbols`

If the object file contains a symbol table chunk (named `OBJ_SYMT`), `Number of Symbols` records the number of symbols in the symbol table.

One of the areas in an object file may be designated as containing the start address of any program which is linked to include the file. If this is the case, the entry address is specified as an `Entry Area Index`, `Entry Offset` pair.

`Entry Area Index`

`Entry Area Index`, in the range 1 to `Number of Areas,` gives the 1-origin index in the following array of area headers of the area containing the entry point.

A value of 0 for `Entry Area Index` signifies that no program entry address is defined by this AOF file.

`Entry Offset`

The entry address is defined to be the base address of the entry area plus `Entry Offset`.

Part two of the AOF header consists of a sequence of area headers. Each area header is five words long, and contains the following word length fields:

`Area Name` Gives the offset of that name in the string table (stored in the `OBJ_STRT` chunk. Each area within an object file must be given a unique name. See *The String Table Chunk (OBJ_STRT)* on page 15-21.

`Attributes and Alignment`

This word contains bit flags that specify the attributes and alignment of the area. The details are given in *Alignment* on page 15-3.

`Area Size` Gives the size of the area in bytes. This value must be a multiple of 4. Unless the `Not Initialised bit` (bit 12) is set in the area attributes (see *Attributes and alignment* on page 15-9), there must be this number of bytes for this area in the `OBJ_AREA` chunk. If the `Not Initialised` bit is set, there must be no initializing bytes for this area in the `OBJ_AREA` chunk.

`Number of Relocations`

Specifies the number of relocation directives that apply to this area (which is equivalent to the number of relocation records following the contents of the area in the `OBJ_AREA` chunk. See *The AREAS chunk (OBJ_AREA)* on page 15-13).

`Base Address`

Is unused unless the area has the absolute attribute. In this case, the field records the base address of the area. In general, giving an area a base address prior to linking will cause problems for the linker and may prevent linking altogether, unless only a single object file is involved.

An unused Base Address is denoted by the value 0.

15.3.1 Attributes and alignment

Each area has a set of attributes encoded in the most significant 24 bits of the `Attributes + Alignment` word. The least significant eight bits of this word encode the alignment of the start of the area as a power of 2 and must have a value between 2 and 32 (this value denotes that the area should start at an address divisible by $2^{alignment}$). Table 15-2 gives a summary of the attributes.

Table 15-2 Area attributes summary

Bit	Mask	Attribute Description
8	0x00000100	Absolute attribute
9	0x00000200	Code attribute
10	0x00000400	Common block definition
11	0x00000800	Common block reference
12	0x00001000	Uninitialized (zero-initialized)
13	0x00002000	Read-only
14	0x00004000	Position independent
15	0x00008000	Debugging tables
		Code areas only
16	0x00010000	Complies with the 32-bit APCS
17	0x00020000	reentrant code
18	0x00040000	Uses extended FP instruction set
19	0x00080000	No software stack checking
20	0x00100000	All relocations are of Thumb code
21	0x00200000	Area may contain ARM halfword instructions
22	0x00400000	Area suitable for ARM/Thumb interworking

Some combinations of attributes are meaningless, for example, read-only and zero-initialized.

The linker orders areas in a generated image in the following order:

- by attributes
- by the (case-significant) lexicographic order of area names
- by position of the containing object module in the link list.

The position in the link list of an object module loaded from a library is not predictable. The precise significance to the linker of area attributes depends on the output being generated.

Bit 8 Encodes the *absolute* attribute and denotes that the area must be placed at its *Base Address*. This bit is not usually set by language processors.

Bit 9 Encodes the *code* attribute:

1 Indicates code in the area.

0 Indicates data in the area.

Bit 10 Specifies that the area is a common definition.

Common areas with the same name are overlaid on each other by the linker. The `Area Size` field of a common definition area defines the size of a common block. All other references to this common block must specify a size which is smaller than or equal to the definition size.

If, in a link step, there is more than one definition of an area with the *common definition* attribute (area of the given name with bit 10 set), each of these areas must have exactly the same contents. If there is no definition of a common area, its size will be the size of the largest common reference to it.

Although common areas conventionally hold data, you can use bit 10 in conjunction with bit 9 to define a common block containing code. This is useful for defining a code area which must be generated in several compilation units, but which should be included in the final image only once.

Bit 11 Defines the area to be a reference to a common block, and precludes the area having initializing data (see *Bit 12*). In effect, bit 11 implies bit 12. If both bits 10 and 11 are set, bit 11 is ignored.

Bit 12 Encodes the *zero-initialized* attribute, specifying that the area has no initializing data in this object file, and that the area contents are missing from the `OBJ_AREA` chunk.

Typically, this attribute is given to large municipalized data areas. When a municipalized area is included in an image, the linker either includes a read-write area of binary zeros of appropriate size, or maps a read-write area of appropriate size that will be zeroed at image startup time. This attribute is incompatible with the read-only attribute (see Bit 13, below).

Whether or not a zero-initialized area is re-zeroed if the image is re-entered is a property of the relevant image format and/or the system on which it will be executed. The definition of AOF neither requires nor precludes re-zeroing.

A combination of bit 10 (common definition) and bit 12 (zero-initialized) has exactly the same meaning as bit 11 (reference to common).

Bit 13 Encodes the *read only* attribute and denotes that the area will not be modified following relocation by the linker. The linker groups read-only areas together so that they may be write-protected at runtime, hardware permitting. Code areas and debugging tables must have this bit set. The setting of this bit is incompatible with the setting of bit 12.

Bit 14 Encodes the *position independent* (PI) attribute, usually only of significance for code areas. Any reference to a memory address from a PI area must be in the form of a link-time-fixed offset from a base register (for example, a pc-relative branch offset).

Bit 15 Encodes the *debugging table* attribute and denotes that the area contains symbolic debugging tables. The linker groups these areas together so they can be accessed as a single continuous chunk at or before runtime (usually, a debugger extracts its debugging tables from the image file prior to starting the debuggee). Usually, debugging tables are read-only and, therefore, have bit 13 set also. In debugging table areas, bit 9 (the *code* attribute) is ignored.

Bits 16-22 encode additional attributes of code areas and must be non-zero only if the area has the code attribute (bit 9) set. Bits 20-22 can be non-zero for data areas.

Bit 16 Encodes the *32-bit PC attribute*, and denotes that code in this area complies with a 32-bit variant of the APCS.

Bit 17 Encodes the *reentrant* attribute, and denotes that code in this area complies with a reentrant variant of the APCS.

Bit 18 When set, denotes that code in this area uses the ARM floating-point instruction set. Specifically, function entry and exit use the `LFM` and `SFM` floating-point save and restore instructions rather than multiple `LDFE`s and `STFE`s. Code with this attribute may not execute on older ARM-based systems.

Bit 19 Encodes the *No Software Stack Check* attribute, denoting that code in this area complies with a variant of the APCS without software stack-limit checking.

Bit 20 Indicates that this area is a Thumb code area.

Bit 21 Indicates that this area may contain ARM halfword instructions. This bit is set by armcc when compiling code for a processor with halfword instructions such as the ARM7TDMI.

Bit 22 Indicates that this area has been compiled to be suitable for ARM/Thumb interworking. See the *ARM Software Development Toolkit User Guide*.

Bits 23 to 31 Are reserved and are set to 0.

15.4 The AREAS chunk (OBJ_AREA)

The AREAs chunk contains the actual area contents, such as code, data, debugging data, together with their associated relocation data. An area is simply a sequence of bytes. The endianness of the words and halfwords within it must agree with that of the containing AOF file. An area layout is:

```
Area 1
Area 1 Relocation
...
Area n
Area n Relocation
```

An area is followed by its associated table of relocation directives (if any). An area is either completely initialized by the values from the file or is initialized to zero, as specified by bit 12 of its area attributes. Both area contents and table of relocation directives are aligned to 4-byte boundaries.

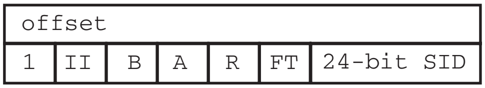
15.5 Relocation directives

A relocation directive describes a value which is computed at link time or load time, but which cannot be fixed when the object module is created.

In the absence of applicable relocation directives, the value of a byte, halfword, word or instruction from the preceding area is exactly the value that will appear in the final image.

A field may be subject to more than one relocation.

Figure 15-1 shows a relocation directive.



offset						
1	II	B	A	R	FT	24-bit SID

Figure 15-1 Relocation directive

`Offset` is the byte offset in the preceding area of the subject field to be relocated by a value calculated as described below.

The interpretation of the 24-bit `SID` field depends on the value of the `A` bit (bit 27):

A=1 The subject field is relocated (as further described below) by the value of the symbol of which `SID` is the zero-origin index in the symbol table chunk.

A=0 The subject field is relocated (as further described below) by the base of the area of which `SID` is the zero-origin index in the array of areas, (or, equivalently, in the array of area headers).

The two-bit field type `FT` (bits 25, 24) describes the subject field:

00 the field to be relocated is a byte.

01 the field to be relocated is a halfword (two bytes).

10 the field to be relocated is a word (four bytes).

11 the field to be relocated is an instruction or instruction sequence.

If bit 0 of the relocation offset is set, this identifies a Thumb instruction sequence, otherwise it is taken to be an ARM instruction sequence.

Bytes, halfwords and instructions may only be relocated by values of small size. Overflow is faulted by the linker.

An ARM branch or branch-with-link instruction is always a suitable subject for a relocation directive of field type *instruction*. For details of other relocatable instruction sequences, refer to 3.6 Handling Relocation Directives on page 3-16.

If the subject field is an instruction sequence, the address in `Offset` points to the first instruction of the sequence, and the `II` field (bits 29 and 30) constrains how many instructions may be modified by this directive:

00	no constraint (the linker may modify as many contiguous instructions as it needs to).
01	the linker will modify at most 1 instruction.
10	the linker will modify at most 2 instructions.
11	the linker will modify at most 3 instructions.

The `R` (pc-relative) bit, modified by the `B` (based) bit, determines how the relocation value is used to modify the subject field:

R (bit 26) = 0 and B (bit 28) = 0

This specifies plain additive relocation. The relocation value is added to the subject field. In pseudo code:

```
subject_field = subject_field + relocation_value
```

R (bit 26) = 1 and B (bit 28) = 0

This specifies pc-relative relocation. To the subject field is added the difference between the relocation value and the base of the area containing the subject field. In pseudo code:

```
subject_field =
subject_field +
(relocation_value-base_of_area_containing(subject_fie
ld))
```

As a special case, if `A` is 0, and the relocation value is specified as the base of the area containing the subject field, it is not added and:

```
subject_field =
subject_field - base_of_area_containing(subject_field)
```

This caters for relocatable pc-relative branches to fixed target addresses.

If `R` is 1, `B` is usually 0. A `B` value of 1 is used to denote that the inter-link-unit value of a branch destination is to be used, rather than the more usual intra-link-unit value.

R (bit 26) = 0 and B (bit 28) = 1

This specifies based area relocation. The relocation value must be an address within a based data area. The subject field is incremented by the difference between this value and the base address of the consolidated based area group (the linker consolidates all areas based on the same base register into a single, contiguous region of the output image).

In pseudo code:

```
subject_field =
subject_field +
(relocation_value -
base_of_area_group_containing(relocation_value))
```

For example, when generating reentrant code, the C compiler places address constants in an address constant area based on register sb, and loads them using sb-relative `LDR` instructions. At link time, separate address constant areas will be merged and sb will no longer point where presumed at compile time. B type relocation of the `LDR` instructions corrects for this.

Bits 29 and 30 of the relocation flags word must be 0. Bit 31 must be 1.

15.6 Symbol Table Chunk Format (OBJ_SYMT)

The `Number of Symbols` field in the fixed part of the AOF header (`OBJ_HEAD` chunk) defines how many entries there are in the symbol table. Each symbol table entry is four words long and contains the following word length fields:

`Name` Is the offset in the string table (in chunk `OBJ_STRT`) of the character string name of the symbol.

`Attributes`

Are summarized in Table 15-2. Refer to *Symbol attributes* on page 15-18 for a full description of the attributes.

`Value` Is meaningful only if the symbol is a defining occurrence (bit 0 of `Attributes` set), or a common symbol (bit 6 of `Attributes` set):

- if the symbol is *absolute* (bits 0-2 of `Attributes` set), this field contains the value of the symbol
- if the symbol is a common symbol (bit 6 of `Attributes` set), this contains the byte length of the referenced common area.
- otherwise, `Value` is interpreted as an offset from the base address of the area named by `Area Name`, which must be an area defined in this object file.

`Area Name` Is meaningful only if the symbol is a non-absolute defining occurrence (bit 0 of `Attributes` set, bit 2 unset). In this case it gives the index into the string table for the name of the area in which the symbol is defined (which must be an area in this object file).

15.6.1 Symbol attributes

Table 15-3 summarizes the symbol attributes.

Table 15-3 Symbol attributes

Bit	Mask	Attribute description
0	0x00000001	Symbol is defined in this file
1	0x00000002	Symbol has a global scope
2	0x00000004	Absolute attribute
3	0x00000008	Case-insensitive attribute
4	0x00000010	Weak attribute
6	0x00000040	Common attribute
		Code symbols only:
8	0x00000100	Code area datum attribute
9	0x00000200	FP args in FP regs attribute
12	0x00001000	Thumb symbol

The Symbol `Attributes` word is interpreted as follows:

Bit 0 Denotes that the symbol is defined in this object file.

Bit 1 Denotes that the symbol has global scope and can be matched by the linker to a similarly named symbol from another object file.

01 Bit 1 unset, bit 0 set. Denotes that the symbol is defined in this object file and has scope limited to this object file (when resolving symbol references, the linker will only match this symbol to references from within the same object file).

10 Bit 1 set, bit 0 unset. Denotes that the symbol is a reference to a symbol defined in another object file. If no defining instance of the symbol is found, the linker attempts to match the name of the symbol to the names of common blocks. If a match is found, it is as if an identically-named symbol of global scope were defined, taking its value from the base address of the common area.

11 Denotes that the symbol is defined in this object file with global scope (when attempting to resolve unresolved references, the linker will match this definition to a reference from another object file).

00 Is reserved.

Bit 2 Encodes the *absolute* attribute which is meaningful only if the symbol is a defining occurrence (bit 0 set). If set, it denotes that the symbol has an absolute value, for example, a constant. If unset, the symbol value is relative to the base address of the area defined by the *Area Name* field of the symbol.

Bit 3 Encodes the *case insensitive reference* attribute which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). If set, the linker will ignore the case of the symbol names it tries to match when attempting to resolve this reference.

Bit 4 Encodes the *weak* attribute which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). It denotes that it is acceptable for the reference to remain unsatisfied and for any fields relocated via it to remain unrelocated. The linker ignores weak references when deciding which members to load from an object library.

Bit 5 Is reserved and must be set to 0.

Bit 6 Encodes the *common* attribute, which is meaningful only if the symbol is an external reference (bits 1, 0 = 10). If set, the symbol is a reference to a common area with the symbol's name. The length of the common area is given by the symbol's *Value* field (see above). The linker treats common symbols much as it treats areas having the *Common Reference* attribute. All symbols with the same name are assigned the same base address, and the length allocated is the maximum of all specified lengths.

If the name of a common symbol matches the name of a common area, these are merged and the symbol identifies the base of the area.

All common symbols for which there is no matching common area (reference or definition) are collected into an anonymous, linker-created, pseudo-area.

Bit 7 Is reserved and must be set to 0.

Bits 8-11 encode additional attributes of symbols defined in code areas.

Bit 8 Encodes the *code datum* attribute which is meaningful only if this symbol defines a location within an area having the *Code* attribute. It denotes that the symbol identifies a (usually read-only) datum, rather than an executable instruction.

Bit 9 Encodes the *floating-point arguments in floating-point registers* attribute. This is meaningful only if the symbol identifies a function entry point. A symbolic reference with this attribute cannot be matched by the linker to a symbol definition which lacks the attribute.

Bit 10 Is reserved and must be set to 0.

Bit 11 Is reserved and must be set to 0.

Bit 12 The Thumb attribute. This is set if the symbol is a Thumb symbol.

15.7 The String Table Chunk (OBJ_STRT)

The string table chunk contains all the print names referred to from the *header* and *symbol table* chunks. This separation is made to factor out the variable length characteristic of print names from the key data structures.

A print name is stored in the string table as a sequence of non-control characters (codes 32-126 and 160-255) terminated by a NULL (0) byte, and is identified by an offset from the start of the table. The first four bytes of the string table contain its length (including the length of its length word), so no valid offset into the table is less than four, and no table has length less than four.

The endianness of the length word must be identical to the endianness of the AOF and chunk files containing it.

15.8 The Identification Chunk (OBJ_IDFN)

This chunk should contain a string of printable characters (codes 10-13 and 32-126) terminated by a NULL (0) byte, which gives information about the name and version of the tool which generated the object file.

Use of codes in the range 128-255 is discouraged, as the interpretation of these values is host-dependent.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.